# Factoring Point Clouds into Positions and Translations for Compression

Patrick Owen
University of Maryland
powen@umiacs.umd.edu

Amitabh Varshney
University of Maryland
varshney@umiacs.umd.edu

## ABSTRACT

Point cloud data is taken from a still shot of a video taken from a Microsoft Kinect. A basic algorithm is proposed for finding frequently occurring translations that map the point cloud to itself. This algorithm is in turn used to generate a set of translations combined with a set of vertices that together cover the original point cloud dataset. Each vertex is associated with a subset of these translations using a bit array. A 45% reduction of the number of vertices to be sent to the graphics card is achieved without reducing the number of points rendered or the apparent quality. Compression of the point cloud is achieved with a compression ratio of 1.47. We hope to increase this number in the future, as this will help reduce the bandwidth necessary to render scenes scanned from the real world.

## 1. INTRODUCTION

It is often useful to be able to view real-world places or events with the ability to move the viewing position interactively. This has become especially true with the rising popularity of virtual reality because it grants users the ability to immerse themselves in a scene. For instance, a 3D reconstruction of an operating room can be used to train surgeons by allowing them to observe a surgery at any angle without being afraid of interfering.

Data scanned from the real world often comes in the form of point clouds, so ideally, point cloud data should be rendered in real time. However, highly detailed scenes often contain animations with millions of points. Without compressing or simplifying the data, we are limited by the maximum transfer rate of data to the graphics card.

However, instead of sending each point to be rendered to the graphics card, one can instead send a set of points and a set of transformations to be applied to each point. This greatly decreases the amount of data that needs to be sent to the graphics card because it is proportional to the number of points *plus* the number of transformations, while the number of points that are actually rendered is the number of points *times* the number of transformations. Unfortunately, most point sets cannot be expressed in that way exactly. However, with modifications to this scheme, some amount of compression can be achieved for real world data.

For instance, data scanned from the real world can often have repeating patterns such as keys on a keyboard or bricks on the side of a building. If the vertices for one component is stored, other components can be drawn by associating each component with one translation. Subtler self-similarities also exist, and the purpose of this compression scheme is to use these self-similarities to build up a list of vertices and transformations to apply.

While any transformation could be used, for the purposes of this paper, we restrict ourselves to translations. Decompression involves taking each position and applying each associated translation to it. If we define this process as "multiplying" the position set by the translation set, then the purpose of the compression scheme is to factor a point cloud into position and translation sets.

## 2. RELATED WORK

Due to the usefulness of compressing point clouds, several schemes have already been invented and evaluated. For instance, point clouds can be converted into octrees to group together the positions of nearby vertices [5]. Julius Kammerl et al. use a modified octree data structure to encode temporal changes, allowing point cloud streams representing an animation to also be compressed [1]. Eduardo Pavez et al. take a different approach by using and compressing a polygon cloud instead of a point cloud [4]. Polygon clouds act as a compromise between a point cloud and a mesh because meshes are easier to render, and point clouds are easier to scan.

For something more similar to our approach, Kim et al. convert point clouds into vertex and transformation streams to improve communication bandwidth [2]. This creates the added advantage that decompression is simply a matter of applying each relevant transformation to each vertex. Their approach consists of discretizing the point set and using the fast Fourier transform to find the autocorrelation of the point cloud. This is in turn used to find frequently occurring translations, which is used to build the transformation streams. Unfortunately, the use of the fast Fourier transform means that the computational complexity grows exponentially with the number of bits of quantization.

Maximo et al. present an algorithm to identify self-similarities in a mesh, which is similar to finding common transformations that map parts of a point cloud to itself, but it focuses more on the benefit such identified similarities has for mesh processing [3], rather than the reduction of communication bandwidth.

## 3. APPROACH

The algorithm being described is designed to convert a point cloud, denoted the *original point cloud*, into a *compressed point cloud*, whose format is described later. To avoid ambiguities, in this paper, a *vertex* is defined to be

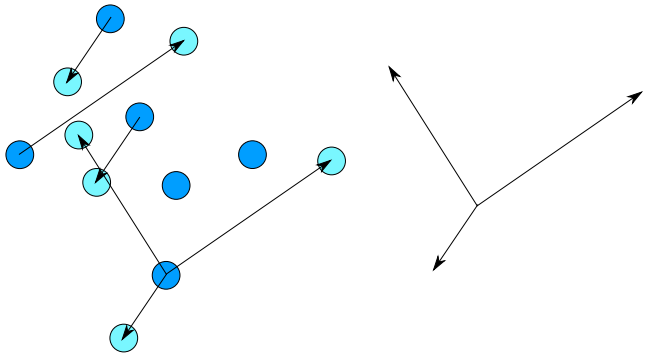

**Figure 1: A visualization of a 2D version of a compressed point could. The darker points represent source vertices, and the lighter points represent virtual vertices. This compressed point cloud has three translations, and each source vertex uses a subset of these translations to create virtual vertices. Source vertices and virtual vertices together form the compressed point cloud.**

a point in a point cloud, and a *point* refers to any point in space. Each vertex has a *vicinity*, which is a region defined to be "close enough" to the vertex so that a point in that region can replace the vertex in the compressed point cloud.

A compressed point cloud is a set of translations and a set of source vertices. Each source vertex is associated with a possibly-empty subset of the translations. To decompress this point cloud, for each source vertex, a vertex is added with the same position, and additional *virtual vertices* are added for each translation by applying the translation to the source vertex. Compression is achieved because all that needs storing are the source vertices, the translations, and a bit array for each source vertex. Each element of the bit array determines whether or not the given source vertex is associated with a given translation. A visualization of a compressed point cloud is shown in Figure 1.

## 3.1 Overview

In this paper, we describe and test an algorithm that converts a point cloud to a compressed point cloud as described previously. This algorithm starts by calling each vertex in the original point cloud a *potential source vertex*. These are vertices that are not associated with any translations, and they may or may not become source vertices in the compressed point cloud by the time the algorithm finishes. The next step is to iteratively add translations. Each translation is associated with some of the vertices, turning potential source vertices into source vertices when necessary, and the resulting virtual vertices are used to remove potential source vertices from the point cloud and act as their replacement. At the end of the algorithm, all remaining potential source vertices are turned into source vertices without any translations.

The main contributor to the size of the compressed point cloud is the number of source vertices. This is true because the number of translations is limited by the size of the bit array associated with each source vertex, so there should be many more source vertices than translations. For this reason, running the compression algorithm should ideally result in as few source vertices as possible, instead relying on

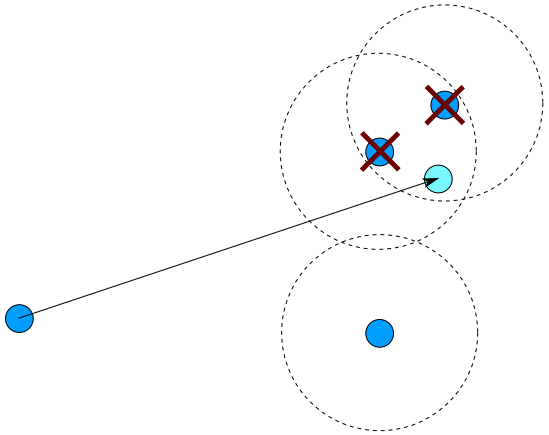

**Figure 2: A virtual vertex removes a given potential source vertex if it in the vicinity of that potential source vertex. In this diagram, a virtual vertex is at a location that allows it to remove two potential source vertices, which are crossed out. The dotted circles are the boundaries of the vicinities of the potential source vertices. Here, vicinity is defined as being a fixed distance $\epsilon$ from the vertex.**

virtual vertices to maintain the quality of the point cloud. In other words, the algorithm should remove as many potential source vertices as possible.

## 3.2 Measuring Closeness

For this approach to work, there must exist translations that map several vertices to other vertices in the same point cloud. However, the coordinates of a vertex are given by floating point values whose precision far exceeds the density of the point cloud, so it is unlikely for such a translation to exist. Therefore, some sacrifices have to be made, and instead of finding translations that map source vertices to other vertices in the point cloud exactly, the correspondence can be approximate. Instead of checking if a point and a vertex are at exactly the same place, one can check if the point is in the vicinity of the vertex. See Figure 2 for an example of how vicinity is used to determine how virtual vertices replace potential source vertices.

A simple way to define vicinity is to declare that a given point is in the vicinity of a given vertex if the distance between them is less than some fixed value $\epsilon$. If $\epsilon$ is too large, the compressed point cloud will differ significantly from the original point cloud when rendered, reducing quality. If $\epsilon$ is too small, then most source vertices will not be associated with any translations, causing the size of the compressed point cloud to be too similar to the size of the original point cloud.

While this approach is simple, it does not take into account the fact that how far a vertex can be shifted without degrading the quality of the point cloud depends on the direction it is being shifted. Point clouds scanned from the real world contain groups of points which are samples of surfaces. Therefore, the exact location of each vertex is less important than the location and shape of the surfaces the vertices represent. In the real world, most surfaces are smooth and locally approximate a plane. Therefore, errors in a direction orthogonal to this plane tend to be worse than errors in a

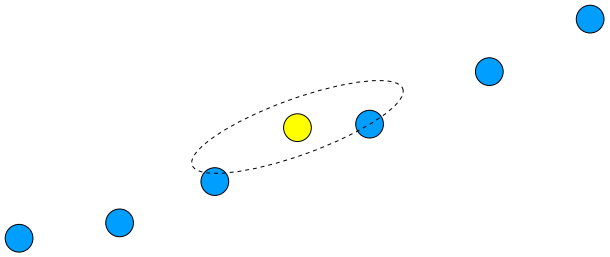

**Figure 3: Under the more complicated definition of vicinity, the shape of the boundary becomes an ellipsoid instead of a sphere, or in the 2D case, an ellipse instead of a circle. The dotted ellipse in this diagram represents the highlighted vertex's vicinity. If it is a potential source vertex, it will be removed if a virtual vertex lands within this ellipse.**

direction parallel to the plane.

To find the plane corresponding to a single vertex and its immediate neighborhood, principal component analysis is applied to the vertices within a given distance from the vertex of interest. The principal component with the smallest variance is considered to be orthogonal to the plane. A plane can then be constructed containing the vertex of interest and orthogonal to that principal component.

Keeping this plane in mind, the vicinity of this vertex is defined to be the interior of an ellipsoid rotated to be aligned to the principal components and translated so that the ellipsoid is centered on the vertex of interest. The radius in each of the three directions corresponding to principal components is proportional to the square root of the variance of that principal component, or proportional to the standard deviation. The scale factor used is configurable but constant for every vertex in the point cloud. To prevent extreme cases for unusual vertex configurations, a configurable minimum and maximum radius, fixed in the same way as the scale factor, is used to clamp each of the three radii of the ellipsoid.

As a special case, if there are not enough vertices close enough to the vertex of interest to run principal component analysis, the vicinity is defined by a sphere centered on the vertex of interest whose radius is the minimum radius. The vast majority of vertices should not fall into this category.

This modification creates an improvement by allowing the compressed point cloud to appear more similar to the original point cloud without affecting the compression ratio. Therefore, this definition of vicinity is used for the remainder of this paper. See Figure 3 for a visualization of this definition of vicinity.

## 3.3 Rating Translations

To achieve a good amount of compression, a translation should allow a large number of potential source vertices to be removed and replaced with virtual vertices. However, it is often difficult to determine how many potential source vertices can be removed by a given translation because a single vertex may be removable and also able to remove another vertex, and a decision would need to be made (See Figure 4).

However, the effectiveness of a translation can be more

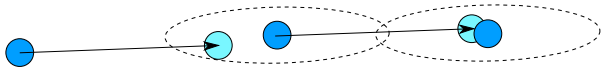

**Figure 4: In this diagram, there are 3 darker points, representing potential source vertices. The lighter points represent the location the virtual vertex would be if its associated potential source vertex became a source vertex. The two rightmost vertices are replaceable with this transformation, but if the rightmost vertex is removed, the middle vertex becomes a source vertex, so it cannot be removed.**

easily estimated. If we assume that every virtual vertex removes exactly one potential source vertex, we can rate a translation by counting how many vertices are sent into the vicinity of a potential source vertex after the translation. This estimation works because the vertices that are included in this count can be used to generate virtual vertices that remove a vertex from the point cloud. For brevity, the count will be called the *coherence* of the translation.

While this is usually a good estimate, the coherence vastly overestimates the effectiveness of a translation when the distance of the translation is too short. In that case, some or all potential source vertices can be translated to the vicinities of themselves. For instance, the identity transformation has a coherence equal to the number of potential source vertices in the point cloud because every point is in the vicinity of itself. To fix this, the only translations considered are those with a distance larger than the configured maximum radius used when defining vicinity. This makes it impossible to map a vertex to the vicinity of itself.

To allow a large number of translations to be tested, instead of translating every vertex to find the exact coherence, we can translate a random sample of vertices to estimate the coherence. To elaborate, we choose a random point out of the pool of source vertices and potential source vertices. Then, we find its corresponding virtual vertex for the translation being tested. If this virtual vertex is in the vicinity of any point in the point cloud, we add 1 to the estimated coherence. The process repeats enough times to create a large enough sample size. Then, the estimated coherence is scaled by a factor equal to the number of vertices divided by the sample size.

## 3.4 Finding Good Translations

One way to discover translations with high coherence is to repeatedly pick a translation between two random vertices that are far enough apart and measure its coherence. The translation with the highest measured coherence can then be chosen. This method is useful because, given all possible translations, the probability that a specific translation will be chosen in a given round is approximately proportional to its coherence. However, many more translations have a low coherence than a high coherence, so it is still unlikely to find a good translation without trying many translations.

Using this method to discover translations has made an interesting trend apparent: smaller translations tend to be better than larger translations. In fact, most small translations have a high coherence. This can be explained by noting that if a vector is on a smooth surface, and it is translated in a direction parallel to the surface at that point with a short enough distance, the resulting virtual vertex is likely to be

in the vicinity of another vertex on that surface.

Given this property, an improved method of finding good translations can be used. It is similar in that it tests multiple translations. However, instead of choosing two random vertices, one random vertex is chosen, and another random vertex within a given fixed configurable distance is then chosen. Finally, the coherence of the translation between them is estimated. This ensures that all tested translations are small, which means that fewer translations will need to be tested to find one with a high coherence.

## 3.5 Choosing Source Vertices

Once a translation is chosen, the next step is to decide which vertices should use that translation. At first, we choose every vertex whose corresponding virtual vertex is in the vicinity of a potential source vertex. Each chosen vertex is associated with the chosen translation. Ideally, this would be the end of the selection process, as each chosen vertex would be able to remove at least one more potential source vertex.

However, the only vertices that can be removed are those without any translations. Since each chosen vertex is associated with a translation, if it was a potential source vertex, it is changed into a source vertex. Consequently, chosen vertices that previously seemed useful may stop being useful. For a given chosen vertex, all the vertices it was going to remove may have turned into source vertices. Since including such vertices for the translation would not help the compression at all, these vertices are disassociated with that translation.

This completes the process of choosing source vertices. It should be noted that vertices that were disassociated may be changed back into potential source vertices if they were not previously associated with any translations beforehand. This creates the opposite situation as before, as newly reestablished potential source vertices could be removable with the selected translation. However, it is assumed that such situations are uncommon enough that they can be ignored because it handling such cases is a matter of optimization rather than correctness.

This method of choosing source vertices for a given translation makes the number of source vertices chosen similar to the number of points that are subsequently removed in the next step. While it may seem like the safeguards of this procedure would guarantee that each source vertex for the translation should remove at least one vertex, this is not always the case. For instance, two chosen vertices may remove the same potential source vertex, creating a redundancy. It is assumed that this situation is uncommon enough to be ignored.

Once the source vertices have been chosen, they are used to remove as many potential source vertices as possible with the chosen transformation. This marks the end of an iteration, and the whole process repeats for the next iteration.

## 3.6 Configurable Parameters

Now that the full algorithm has been described, it is helpful to review its parameters, as the power and quality of the compression depend on these parameters.

As the algorithm loops, a parameter is needed to refer to the number of times it loops. This is referred to as the number of *iterations*. Each iteration adds a single translation to the compressed point cloud, so the number of translations at the end is equal to the number of iterations.

**Figure 5: A visualization of the point cloud dataset from a point of view similar to the location of the Kinect that scanned the image.**

Another parameter is how many translations to try before picking the one with the highest estimated coherence. This is referred to as the number of *trials*. Another parameter is how many samples to take when estimating the coherence of a transformation. This is called the number of *trial samples*. The range of magnitudes allowed for each tested translation is configurable, and it is referred to as the *minimum* and *maximum translation*.

A few more parameters are needed to determine how to find the vicinity of each vertex. Each radius of the ellipsoid is bounded above and below, and these bounds are referred to as the *minimum* and *maximum vicinity radius*. Principal component analysis is applied to all vertices within a given distance to the vertex of interest, and this given distance is called the *vicinity search radius*. The square root of each variance found is then multiplied by a configurable constant, and this is denoted the *vicinity scale factor*.

The minimum translation is always set to be equal to the maximum vicinity radius, as the only purpose of the minimum translation is to prevent any vertex from being mapped to the vicinity of itself.

# 4. RESULTS

This algorithm has been tested on a point cloud obtained from a Kinect recording a surgical operation. The parameters for this algorithm have effects on the power and quality of the compression. Compression ratios of approximately 1.5 have been achieved, but, due to an inherent limitation in the algorithm, no combination of parameters has been able to increase the compression ratio much further. The effect of each parameter on the compression is discussed.

The main factor that determines the power of the compression algorithm is how many vertices are removed to be replaced with virtual vertices. Therefore, the number of vertices removed is used to estimate the relative power of the compression algorithm.

## 4.1 The Dataset

The point cloud used to analyze the compression algorithm is shown in Figure 5. It consists of 178,774 points. As the data comes from a single Kinect, each person and object leaves a shadow in which no points are visible. In addition, points on the back wall are more spread out than

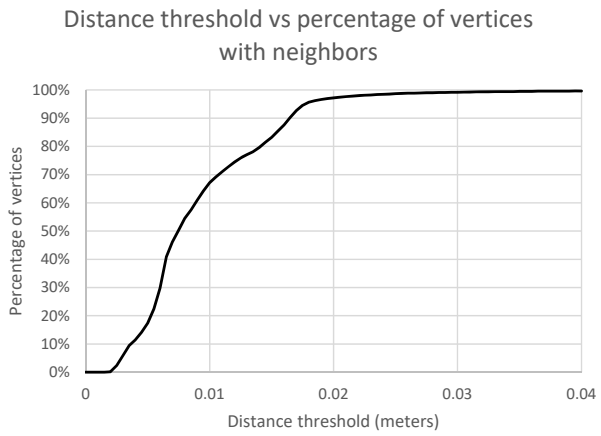


**Figure 6: The relationship between a chosen distance threshold and the number of points with at least one other point within that threshold. Note that above 0.02 meters, relatively few isolated vertices are left.**

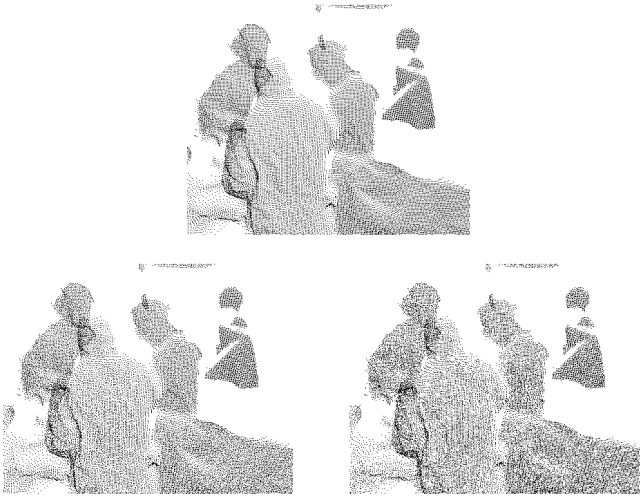

**Figure 7: At the top is a section of the point cloud before compression. At the bottom left is the compressed point cloud with the chosen parameters. At the bottom right is the compressed point cloud with all parameters representing an absolute distance doubled (minimum and maximum vicinity radius, vicinity search radius, and minimum and maximum translation). If viewed on a computer monitor, it may be necessary to zoom in. Note that the bottom-right point cloud has visible gaps rather than a relatively uniform distribution of points.**

points closer to the camera. The compression algorithm uses vicinity to describe if points are close enough together to act as substitutes for each other, but how close a point needs to be to fulfill this requirement depends on the density of the point cloud. The denser a point cloud is, the more precise the location of each point needs to be, shrinking the region that can be defined as the vicinity. A basic way of estimating the point cloud density is to count how many vertices have a neighbor within a given distance threshold, and to see how changing this threshold affects the count. Figure 6 shows the result for the point cloud of interest.

## 4.2 Parameters Used

For point clouds of uniform density, the definition of vicinity should ideally be that few vertices are too deep in the vicinity of other vertices. Otherwise, replacing vertices with virtual vertices may reduce the density of the point cloud at certain locations, creating visible gaps. Therefore, the maximum vicinity radius should not be greater than 0.02 because more than 95% of points have neighbors within that distance (See Figure 6). To test the algorithm, we use a maximum vicinity radius of 0.01. The minimum vicinity radius is set to 0.002 because visual inspection has revealed that displacing vertices in any direction with a distance less than this amount does not noticeably affect the quality. See Figure 7 for reference.

The vicinity search radius is set to 0.02, double the maximum vicinity radius. A vicinity scale factor of 0.5 is used to compromise for the fact that the search radius is double the maximum radius. The minimum transformation is set to 0.01, preventing a point from being translated into its own vicinity, and the maximum is set to 0.02, double the minimum.

We use 1000 trial samples for estimating coherence because this number is large enough to provide a good estimate of the coherence, and it is small enough to allow trials to be run quickly.

## 4.3 Accuracy of Coherence

The algorithm uses a measure of coherence of a transformation, which is an estimation for the number of vertices it will remove. To measure how good of an estimation this is, 500 random transformations were selected using the same selection scheme as the one the algorithm uses. Figure 8 is a scatterplot comparing the coherence of the transformation to the number of vertices it removes when used for a single iteration of the algorithm.

As shown in Figure 8, the coherence overestimates the number of vertices removed by approximately 30%. For instance, for transformations with a coherence of around 16%, only 12% of the vertices are removed.

As for estimated coherence, it is important to quantify how accurate the estimate is. If the amount of vertices included in the actual coherence relative to the total number of vertices is $p$, then for each sample, the probability that the randomly chosen vertex will be included in the coherence estimation is $p$, so for $n$ samples, the possible values for the estimated coherence before scaling falls in a binomial distribution with $n$ trials and probability $p$. For 1000 samples, in the worst case that $p = 0.5$, the difference between the coherence and the estimated coherence will be less than 3.2% of the total number of vertices 95% of the time.

## 4.4 Trials and Iterations

We finally examine how the number of iterations and the number of trials affects the number of vertices removed. The algorithm has been tested with 1 to 32 iterations and 1, 10, and 100 trials. Because the algorithm uses randomness, it has been tested 30 times. The results are shown in Figure 9.

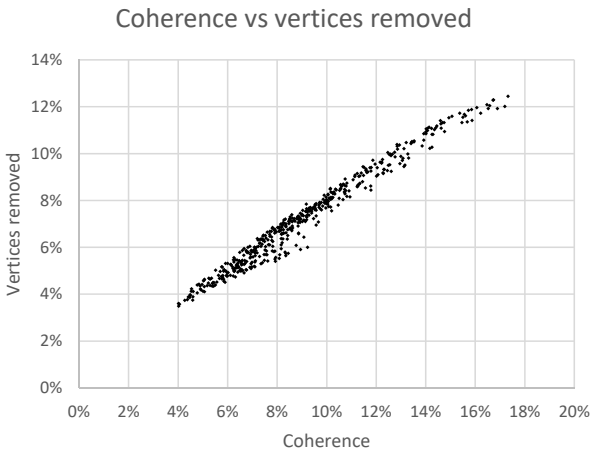


**Figure 8: A plot showing the measured coherence and actual effectiveness of 500 randomly chosen transformations relative to the number of points in the point cloud. A strong positive correlation is apparent, as expected, but coherence tends to overestimate the effectiveness of the transformation by about 30% to 40%.**

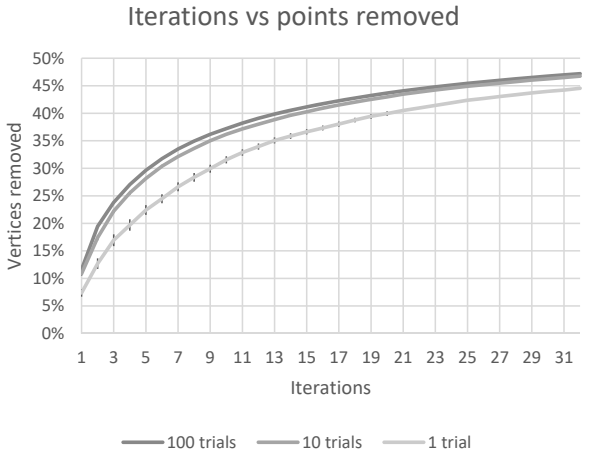


**Figure 9: A plot showing how the number of iterations and trials affects the number of vertices removed. Error bars for a 95% confidence interval are included.**

The first thing to note is that the first iteration removes the most points, and future iterations become much less effective. When 100 trials are used, a 25% reduction in point count is achieved by the fourth iteration. A 30% reduction is achieved by the sixth iteration. However, it takes 14 iterations to achieve a 40% reduction and 24 iterations to achieve 45%.

Another thing to note is that despite the order of magnitude differences in the number of trials, the effect on the point count reduction is less pronounced, especially after 14 iterations, when the reduction achieved when using 100 trials is only 5% more than the reduction achieved when using 1 trial. This similarity can be explained in two different ways.

Firstly, the distribution of coherences of transformations from one vertex to a nearby vertex is not heavily skewed (See Figure 8). This means that it does not take many tries to find a transformation with a coherence reasonably close to that of the best transformation.

Another reason is choosing a transformation with a high coherence in one iteration reduces the number of points that can be removed in future iterations, which in turn decreases the effectiveness of future transformations because many potential source vertices have been turned into source vertices. For example, if each source vertex is associated with one translation that removes a single vertex, then at most a 50% reduction can be achieved regardless of the number of trials. In fact, as shown in Figure 9, the number of vertices removed appears to asymptotically approach 50%.

## 4.5 Compression Ratio

The compression ratio of this algorithm depends on how much data is needed to express the location of a single vertex in the original point cloud. For instance, suppose each vertex is represented as three 32-bit floating point number, one for each coordinate, yielding a total of 96 bits. Then, if 32 translations are used, an additional 32 bits are added to each vertex, yielding a total of 128 bits per vertex.

As shown in Figure 9, for the given example point cloud, 47% of the vertices are removed, but each vertex holds 33% more data. Consequently, the compression ratio is approximately 1.42. If 16 translations are used instead, only 16 extra bits are needed for each vertex. Although the number of vertices removed is reduced to 42%, the compression ratio increases to 1.47. With 8 translations, the compression ratio is back down to 1.42, which means that using 16 translations is ideal for this dataset, as the associated bit arrays can represented as 16-bit integers.

However, if each coordinate is represented using just 16 bits instead of 32 bits, the ideal number of translations is reduced to 8, and the compression ratio is reduced to 1.32.

# 5. CONCLUSION

The algorithm presented is successfully able to compress a point cloud into a set of translations and a set of source points, each source point associated with a subset of the translations. A major advantage of this scheme is that decompression is simple and easy to parallelize because the procedure is simply to apply each relevant translation to each source point and output the result. Each source vertex can be treated independently this way.

Currently, the number of source vertices in the compressed point cloud can reach 50% of the number of original ver-

tices. Reducing this number any further is not possible with the current implementation because no source points can be removed without removing virtual vertices unintentionally and leaving a visible gap in the point cloud. While a single source point can be associated with many transformations, in practice, most source vertices are associated with only one translation. This means that when half of the potential source vertices are removed, most of the other half has turned into source vertices, creating the limitation.

The algorithm presented is just one of many ways to approach compressing a point cloud. It has the advantage of not reducing the complexity of the point cloud while being easy to decompress. A disadvantage is that each source vertex needs to be associated with a bit array identifying its translations, and the compression ratio is limited to 2:1 at the most, often being much smaller. It is also a lossy compression scheme, as vertices are effectively moved to some point in their vicinity.

It should be possible to achieve better compression by tweaking the compression algorithm to overcome the 50% limitation.

## 5.1 Future Work

The next steps for this algorithm would be to improve its compression ratio. It should be possible to achieve better compression by tweaking the compression algorithm to increase the likelihood for a single source vertex to be associated with more than one translation. Limiting the number of source vertices could allow this to happen.

For instance, the algorithm turns as many potential source vertices as possible into source vertices in each iteration. There is a cost involved in turning a potential source vertex into a source vertex because the resulting source vertex can never be removed. Instead, reusing existing source vertices for multiple translations is desirable. Redefining coherence to favor translations from existing source vertices and being more conservative about converting potential source vertices into source vertices should make it possible to achieve better compression ratios.

Another potential improvement is to make the algorithm hierarchical. The resulting source vertices after one pass of the algorithm can be used as the original point cloud in another pass of the algorithm. It is unclear whether this approach is realistic, as the bit array of each source vertex has to be conserved even when turned into a virtual vertex by the second pass of the algorithm.

A major limitation of the algorithm is that it completely ignores any data associated with each vertex other than the position. As point clouds in the real world have color information, another scheme would need to be added on top of the algorithm in this paper to allow the reconstruction of this color data.

Lastly, this compression scheme may be able to be combined with other existing point cloud compression schemes to improve upon them. For instance, running this algorithm after reducing the number of points in the point cloud is trivial. Combining it with compression schemes that are more complicated than reducing the number of points is not guaranteed to create an improvement, as the factoring algorithm is lossy, and a bit array must be associated with each vertex. It can improve octree compression because instead of encoding every vertex, only source vertices need to be encoded. The associated bit array can be included in the same way color information would be included [5].

# 6. ACKNOWLEDGEMENTS

We would like to thank everyone at the Graphics and Visual Informatics Laboratory (GVIL), including Ruofei Du, Eric Krokos, Eric Lee, Mukul Agarwal, and others, for providing feedback and answering any questions we had. We would also like to thank Mukul Agarwal for providing us access to the point cloud dataset and Ruofei Du for finding the paper about polygon cloud compression.

# 7. REFERENCES

[1] J. Kammerl, N. Blodow, R. B. Rusu, S. Gedikli, M. Beetz, and E. Steinbach. Real-time compression of point cloud streams. *2012 IEEE International Conference on Robotics and Automation*, May 2012.

[2] Y. Kim, C. H. Lee, and A. Varshney. Vertex transformation streams. *Graphical Models*, 68(4):371–383, July 2006.

[3] A. Maximo, R. Patro, A. Varshney, and R. Farias. A robust and rotationally invariant local surface descriptor with applications to non-local mesh processing. *Graphical Models*, 73(5):231–242, September 2011.

[4] E. Pavez, P. A. Chau, R. L. de Queiroz, and A. Ortega. Dynamic polygon cloud compression. *arXiv:1610.00402*, October 2016.

[5] R. Schnabel and R. Klein. Octree-based point-cloud compression. *SPBG'06 Proceedings of the 3rd Eurographics / IEEE VGTC conference on Point-Based Graphics*, pages 111–121, July 2006.