Lossless Geometry Compression for Steady-State and Time-Varying Irregular Grids

SUBMISSION ID: paper1275

Abstract
In this paper we investigate the problem of lossless geometry compression of irregular-grid volume data represented as a tetrahedral mesh. By geometry compression we mean compression of vertex coordinates and the associated scalar values. We propose a novel lossless compression technique that effectively predicts, models, and encodes the geometry data, for both steady-state (i.e., with only a single time step) and time-varying datasets. It can be used with low precision quantized data or with high precision unquantized data. It can also be easily integrated with a class of the best existing connectivity compression techniques for tetrahedral meshes with a small amount of overhead information, which pays off by the huge gains obtained in our geometry compression. Our geometry coder does not need any connectivity information, and involves several interesting ideas such as kd-tree-like partitioning/clustering, traveling salesperson problem (TSP) modeling, and a two-layer modified Huffman coding technique based on an optimal alphabet partitioning approach using dynamic programming or greedy heuristic. The experiments show that our technique achieves superior compression ratios, with reasonable encoding times and very fast (linear) decoding times. Our technique also exhibits a nice trade-off between compression ratios and encoding times. Even the lower complexity versions of our technique perform significantly better than the state-of-the-art approach in geometry compression of irregular-grid volume data.

1. Introduction

In the past several years, new challenges for scientific visualization have emerged as the size of data generated from the simulations has grown exponentially [BKC97]. The emerging demand for efficiently storing, transmitting, and visualizing such data in networked environments has motivated graphics compression for 3D polygonal models and volumetric datasets to become a focus of research in the past several years. For volumetric data, the most general class is irregular-grid volume data represented as a tetrahedral mesh. It has been proposed as an effective means of representing disparate field data that arise in a broad spectrum of scientific applications including structural mechanics, computational fluid dynamics, partial differential equation solvers, and shock physics.

Although there has been a significant amount of research done on graphics compression, most techniques reported in the literature have mainly focused on compressing the connectivity information, rather than the geometry information which consists of vertex-coordinates and data attributes (such as scalar values in our case). As a result, while connectivity compression already achieves an impressive compression rate of 1–2 bits per triangle for triangle meshes [TR98, Ros99, AD01, TG98] and 2.04–2.31 bits per tetrahedron for tetrahedral meshes [GGS99, YMC00], progress made in compressing the geometry information has not been equally impressive. For a tetrahedral mesh, typically about 30 bits per vertex (not including the scalar values) are required after compression [GGS99], and we do not know of any reported results on compressing time-varying fields over irregular grids (see Section 2). Given that the number of cells is about 4.5 times the number of vertices in typical tetrahedral meshes and that the connectivity compression ratio is about 2 bits per cell, it is clear that geometry compression is by far the dominating bottleneck. The situation gets much worse for time-varying datasets where each vertex can have hundreds or even thousands of time-step scalar values. Clearly, more attention has to be paid to geometry compression if we hope to obtain a significant improvement in the overall compression efficiency.

The disparity between bit rates needed for representing the connectivity and the geometry gets further amplified when lossless compression of geometry information is required. Interestingly, whereas almost all connectivity compression techniques are lossless, the geometry compression
results in the literature almost always include a quantization step which makes them lossy (the only exception is the very recent technique of [ILS04] for polygonal meshes). Since the scalar values are associated with the vertices, the resulting compression of the scalar values is also lossy. While lossy compression might be acceptable for usual graphics models to produce an approximate visual effect to “fool the eyes,” it is of serious concern for scientific datasets where accuracy is of vital importance. This is especially true for applications such as medical diagnosis, simulation, analysis of structural properties, and so on. These applications would usually require truly lossless techniques for geometry compression.

In this paper we develop lossless geometry compression techniques for tetrahedral volume data, for both steady-state and time-varying datasets. We take a novel direction in that our geometry coder is independent of connectivity coder (albeit they can be easily incorporated) and may re-order the vertex list differently from connectivity-coder traversal. This direction has not been attempted before since it typically incurs an additional overhead of $\log_2 n$ bits per vertex (b/v)—generally considered very expensive—for recording the permutation of the $n$ vertices in the mesh when incorporated with a connectivity coder. Our rationale for taking this direction is twofold. First, since geometry compression is the dominating bottleneck, we want to see how far we can push if the geometry coder is not “burdened” by connectivity compression. Secondly, such geometry coder is interesting in its own right for compressing datasets not involving any connectivity information, such as point clouds, which are becoming increasingly important nowadays. It turns out that our geometry coder significantly improves the state-of-the-art techniques even after paying the extra $\log_2 n$ b/v. Moreover, we show that the permutation sequence can be efficiently encoded so that the extra overhead is far less than $\log_2 n$ b/v, resulting in even greater improvements in our overall compression rates.

Our method makes use of several interesting ideas. We formulate the problem of optimally re-ordering the vertex list as a combinatorial optimization problem, namely, the traveling salesman problem (TSP), and solve it with heuristic algorithms. To obtain a much better computation efficiency, we first perform a $kd$-tree-like partitioning/clustering, and then re-order the vertices within each cluster by solving the TSP problem. Our coding technique is a two-layer modified Huffman coding based on an optimal alphabet partitioning approach using dynamic programming or greedy heuristic, where we also combine the data sequences across all clusters to greatly reduce the overall Huffman-table size. It should be noted that although our technique is for lossless compression, it can be used in lossy compression as well where the only additional step is the quantization of vertex coordinates and data values. Our geometry compression can also be easily integrated with a class of the best existing connectivity compression techniques for tetrahedral meshes [GGS99, YMC00] with a small amount of overhead, which pays off for the huge gains we obtain in geometry compression.

The experiments show that our technique achieves superior compression ratios, with reasonable encoding times and very fast (linear) decoding times. Our technique also exhibits a nice trade-off between compression ratios and encoding times. Compared with the state-of-the-art flipping algorithm (extended from the one for triangle meshes [TG98] to tetrahedral meshes), when both integrated with the same state-of-the-art connectivity coder [YMC00], our approach, after paying the connectivity-integration overhead, achieves significant improvements of up to 46.70 b/v (61.2%) for steady-state data, and up to 83.56 b/v (28.9%) for time-varying data.

2. Previous Related Work

There has been a large amount of work on compressing polygonal meshes (e.g., [De95, GS98, KG00, LAD02]). Much of this work has mainly focussed on compressing connectivity information. Compression techniques for polyhedral volume meshes have also been widely studied [SR99, GGS99, PRS99, YMC00]; again the main focus has been on connectivity compression. As mentioned before, these techniques achieve an impressive compression performance of 1–2 bits per triangle for triangle meshes [TR98, Kos99, AD01], and 2.04–2.31 bits per tetrahedron for tetrahedral meshes [GGS99, YMC00].

There are relatively fewer results that focus on compressing the geometry information. Lee et. al. [LAD02] proposed the angle analyzer approach for traversing and encoding polygonal meshes consisting of triangles and quads. Devillers and Gandoin [DG00, GD02] proposed techniques that are driven by the geometry information, for both triangle meshes and tetrahedral meshes. They only consider compressing the vertex coordinates but not the scalar values for the case of tetrahedral meshes.

The most popular technique for geometry compression of polygonal meshes is the flipping algorithm using the parallelogram rule introduced by Touma and Gotsman [TG98]. Isenburg and Alliez [IA02b] extended the parallelogram rule so that it works well for polygonal surfaces beyond triangle meshes. Isenburg and Gumhold [IG03] applied the parallelogram rule in their out-of-core compression algorithm for polygonal meshes larger than main memory. Other extensions of the flipping approach for polygonal meshes include the work in [KG02, CCMW05]. For volume compression, Isenburg and Alliez [IA02a] extended the flipping idea to hexahedral volume meshes. The basic flipping approach of [TG98] can also be extended to tetrahedral meshes, which, combined with the best connectivity coder [GGS99, YMC00], is considered as a state-of-the-art geometry compression technique for tetrahedral meshes. We show in Section 4 that our new algorithm achieves significant improvements over this approach.
We remark that all the previous approaches mentioned above perform vertex-coordinate quantization (to 12 bits per coordinate typically) as the first step and hence are lossy. Although they can be arguably used without the initial quantization, these techniques do not have an appropriate and efficient mechanism for encoding high precision floating-point numbers that we have prior to quantization, and cannot be used directly for truly lossless compression. Very recently, Isenburg et. al. [ILS04] focused on removing the need of the initial quantization, for polygonal mesh compression using the flipping prediction. Our method provides a simple alternative for truly lossless geometry compression as a by-product, and does not need the connectivity information.

There are other compression techniques for regular-grid volume data [SBS02, ILRS03, SW03, BCF03]. All these approaches do not consider compressing the vertex coordinates, since such information is not needed for regular grids.

3. Our Approach

In this section we develop the main ideas behind our compression technique. We first give an overview, and then proceed to describe in detail the individual steps. Finally, we show how to integrate our geometry coder with state-of-the-art connectivity coding algorithms to obtain a complete lossless compression technique.

In the following, we assume that each vertex entry $v_i$ in the vertex list consists of a $t + 3$ tuple $(x_i, y_i, z_i, f_{i1}, ... , f_{it})$ where $x_i$, $y_i$, $z_i$ are the coordinates and $f_{i1}, ..., f_{it}$ are the scalar values for $t$ time steps, with steady-state data ($t=1$) being just a special case. From henceforth, unless otherwise stated, when we say vertex we mean the entire $t + 3$ tuple of coordinate and scalar values. We also assume that each (tetrahedral) cell entry $c_i$ in the cell list has four indices to the vertex list identifying the vertices of $c_i$.

3.1. Overview of Proposed Technique

We adopt a common two-step modeling process for lossless compression [Ris84]. First, we try to capture the underlying structure in the data with a simple-to-describe prediction model. Then, we try to model and encode the prediction residuals.

For the first task of finding a prediction scheme, we observe that irregular-grid volume data is often created from randomly selected points, hence even the popular flipping technique presented in [TG98] that works well with surface data fails to do a good job when applied to irregular-grid data (see Section 4). However, these randomly selected points are typically densely packed in space. Every point has a few other points that are in its immediate spatial proximity. Hence one way to efficiently represent a vertex would be in terms of its difference with respect to some neighboring vertex. This is called differential coding. Differential coding can be seen as a special case of predictive coding where the previous vertex is used as prediction of the current vertex. If vertices are close to one another spatially then clearly the difference between their coordinates will be small and can then be encoded efficiently. Similarly, for scalar values at each time step, if the underlying scalar function is reasonably smooth, then vertices that are close to each other will have scalar values of small differences.

If we are free from the vertex-traversal order imposed by connectivity coder, then clearly there is no need to use a fixed order of vertex list, and different order will lead to different compression performance when using differential coding. We formulate the problem of how to re-order the vertex list to get maximum compression as a combinatorial optimization problem, which is known to be intractable, and hence we propose some simple heuristic solutions. To speed up the computation, we partition the vertices into clusters and then re-order the vertices within each cluster. As mentioned before, vertex re-ordering causes some extra overhead when we integrate with the connectivity coder, and we address the issue of how to reduce this overhead in Section 3.5.

For the second task of encoding the prediction errors/residuals, it is well known that (lossless) text compression techniques like gzip would be ineffective since geometry data is inherently multidimensional and usually very noisy. Another simple approach would be to use a Huffman code. The problem is that the difference values we need to encode are taken from a very large alphabet (some fixed precision real numbers) and the corresponding Huffman table would be so large that it would offset any gain made by the coding technique itself. To address this problem the common approach applied in the literature has been to Huffman code individual bytes of the differences. However, individual bytes of a difference value are highly correlated and such a coding strategy fails to capture these correlations. We use a two-layer modified Huffman code to solve this problem in a simple and effective way.

In summary, our technique for lossless geometry compression consists of the following stages. It works for both steady-state and time-varying irregular grids in essentially the same way.

1. Step 1. Partition vertices into clusters. We first partition the vertices (and their scalar values) into $K$ disjoint clusters (where $K$ is an adjustable parameter) that may contain a few hundred vertices. Clustering is done such that vertices in each cluster are in close spatial proximity of one another.
2. Step 2. Map to integers. The vertex coordinates and scalar values are mapped to integers by an invertible scalar coding strategy that we present in Section 4. This coding strategy maps points to integers by an invertible scalar coding technique that we present in Section 4.
3. Step 3. Re-order the vertices by formulating and solving a TSP problem. For each cluster, we re-order the ver-
3.2. Step 3: Vertex List Re-ordering

As described above, the efficiency of differential coding of the vertex list depends on the way this list is ordered. We thus would like to compute an ordering that minimizes the cost of representing the list with successive differences. More precisely, given an unordered list of vertices \( \{v_1, v_2, \ldots, v_n\} \), we want to compute a permutation \( \pi \) such that the objective function

\[
\sum_{i=2}^{n} C(|v_{\pi(i)} - v_{\pi(i-1)}|)
\]

is minimized. In Equation (1), the function \( C(\cdot) \) represents the cost in bits of representing the difference of two adjacent vertices. For arriving at a cost function we assume that the differences are iid, namely, independently drawn from an identical distribution. However, this does not suffice, as now the cost of representing the differences, which is the zero-order entropy of the differences, itself depends on the frequencies of the differences which in turn vary from one permutation to another. Hence we make another simplifying assumption that the differences are taken from a double sided, zero mean, geometric distribution, and in this case the cost of representing the value \( n \) is simply proportional to \( \log n \). This allows us to restate the problem in Equation (1) as the following traveling salesperson problem (TSP):

Form a complete graph \( G \) where the nodes are the vertex entries and the length of each edge between two entries is the bit length needed to represent their difference under the chosen compression method. Find a Hamiltonian path on \( G \) that visits every node exactly once so that the total path length is minimized.

With the above assumptions, we define the edge length between two nodes \( v_i = (x_i, y_i, z_i, f_{i1}, f_{i2}, \ldots, f_{i_d}) \) and \( v_j = (x_j, y_j, z_j, f_{j1}, f_{j2}, \ldots, f_{jd}) \) to be \( \lg |x_i - x_j| + \lg |y_i - y_j| + \lg |z_i - z_j| + \lg |f_{i1} - f_{j1}| + \cdots + \lg |f_{id} - f_{jd}| \).

Unfortunately TSP is in general an NP-complete problem. However, there are many well-known heuristics to get good solutions for a given instance. In our implementation we chose to use the Simulated Annealing (SA) based heuristics and the Minimum Spanning Tree (MST) based approximation algorithm. The MST algorithm first computes a minimum spanning tree of \( G \), and then visits each node exactly once by a depth-first-search traversal of the tree. This algorithm produces a Hamiltonian path with the total path length no more than twice the optimal path length (i.e., with an approximation factor of 2) if the distances in \( G \) obey a triangle inequality [CLRS01]. Although the triangle inequality may not always hold in our case, we observed that this algorithm did produce comparable-quality solutions and ran much faster \( O(n^2) \) time for \( n \) vertices since \( G \) is a complete graph) than simulated annealing; see Section 4 for details.

We remark that the re-ordering process has to be done just once at compression time and hence the running-time cost is absorbed in the preprocessing stage. The decompression process does not have to perform this re-ordering and can remain computationally very efficient.

3.3. Step 1: Partitioning the Vertex List into Clusters

Although the re-ordering process occurs only once during encoding and never during decoding, the number \( n \) of vertices present in a typical tetrahedral mesh is very large (e.g., hundreds of thousands), making even the \( O(n^2) \)-time MST heuristic infeasible. In fact just computing the weights on the graph \( G \) will need \( O(n^2) \) time since \( G \) is a complete graph. Hence we first partition the vertex list into small clusters and then within each cluster we re-order the vertices by solving the TSP problem.
To achieve this we propose the following simple and effective \( k \)-\( d \)-tree-like partition scheme. Suppose we want to form \( K \) clusters of equal size. We sort all vertices by the \( x \)-values, and split them into \( K^{1/3} \) groups of the same size. Then, for each group we sort the vertices by the \( y \)-values and again split them equally into \( K^{1/3} \) groups. Finally we repeat the process by the \( z \)-values. Each resulting group is a cluster of the same size, and the vertices in the same group are spatially close. In this way, the original running time of \( O(n^2) \) for re-ordering is reduced to \( O(K(n/K)^2) = O(n^2/K) \), a speed-up of factor \( K \). Also, the overall clustering operation takes only \( O(n \log n) \) time.

3.4. Step 5: Encoding Vertex Differences

Entropy coding techniques, such as Huffman coding and arithmetic coding, have been widely applied in the graphics compression literature. Unfortunately, as mentioned in Section 3.1, the vertex differences have a very high dynamic range, namely, they have a large alphabet size, and cannot be efficiently encoded in a straightforward manner. For example, a Huffman code for the entire alphabet would require an unduly large code table. This is the case even if only the codewords for the symbols that occur in the data need to be stored. Such a large Huffman table would offset any efficiency made by Huffman coding. The problem is more than just about table size. Even if one were to use some universal coding technique like adaptive arithmetic coding, the sparsity of samples does not permit reliable probability estimation of different symbols, leading to poor coding efficiency. This problem is known as the "model cost" problem in the data compression literature.

One way to deal with the problem of entropy coding of large alphabets is to partition the alphabet into smaller sets and use a product code architecture, where a symbol is identified by a set number and then by the element number within the set. If the elements to be coded are integers, as in our case, then these sets would be intervals of integers and an element within the interval can be represented by an offset from the start of the interval. The integers are then typically represented by a Huffman code of the interval identifier and then by a fixed-length encoding of the offset. This strategy is called modified Huffman coding and has been employed in many data compression standards in the context of Huffman coding, including the JPEG standard [PM93], the CCITT Group 3 and Group 4 facsimile compression standards [HR80], and the MPEG video compression standards [MPE89].

In fact, one popular way of Huffman coding high-precision numbers of scientific applications applies some sort of partitioning scheme. For example, the 32-bit floating-point numbers to be encoded are viewed as a sequence of four bytes and separate entropy codes are designed for each of the four bytes. This clearly can be viewed as a recursive alphabet partitioning. The first level partitions the numbers into 256 different sets and the elements within each set are further partitioned into 256 sets and so on. Similar comments apply if the numbers are 64 bits or 80 bits or any other representation and if the partitioning is done based on bytes or nibbles or 16-bit chunks, etc.

However, the above approaches for modified Huffman coding are ad-hoc and clearly sub-optimal. In the work of Chen et. al. [CCMW03], the problem of optimal alphabet partitioning for the design of a two-layer modified Huffman code was formulated and a solution was given based on dynamic programming. However, the complexity of the dynamic programming method can be quite prohibitive for a long sequence and a large alphabet. Hence a greedy heuristic was also proposed, which runs in \( O(N \log N) \) time where \( N \) is the number of distinct symbols in the source sequence.

We now briefly review the greedy algorithm of [CCMW03]. We first sort the distinct symbols to be encoded in increasing order. Then we make a linear scan of the symbols starting from the smallest one (or equivalently the largest one). We assign this symbol to a partition in which it now is the only element. We then consider the next smallest symbol, for which we have two choices. The first choice is to include this symbol into the current partition, increasing its cardinality by one. The second choice is to create a new partition for this symbol, thereby closing the current partition. We compare the "costs" of these two choices and use a greedy strategy to select the one with a lower cost. We continue in this manner, visiting each new distinct symbol and assigning it either to the current partition or to a new partition of its own. We refer to [CCMW03] for more details.

In this paper we use the greedy algorithm described above to construct a two-layer modified Huffman code to encode vertex differences. Since we encode multiple clusters and multiple vertex components, we can either use a single Huffman code for each coordinate value (and scalar value) for each cluster, or we can use a single Huffman code for all the data values in all clusters. There are other possible options that are in between these two extreme approaches. In practice we found that the vertex differences for the coordinate values had similar distributions and using a single Huffman code for them gave the best performance, since the overall Huffman-table size can be greatly reduced. The same was the case for the scalar values. Hence we only use two Huffman codes for steady-state data, one for all coordinates across all clusters, the other for all scalar values across all clusters. We call this the Combined Greedy (CGreedy) approach. For the scalar values in time-varying datasets, we use one Huffman code for every \( i \) time steps across all clusters; this is the CGreedyi approach. To take advantage of the similarity between scalar values of adjacent time steps, we also propose another option of CGreedyi: we first perform a differential coding along the direction of the time steps for each vertex, and then let the resulting tuple play the role of

submitted to EUROGRAPHICS 2005.
the original vertex. In summary, our proposed coding methods are CGreedy for steady-state data, and two options of CGreedy for time-varying data. We refer to Section 4 for more detailed descriptions of them.

3.5. Integration with Connectivity Compression

In this section, we show how our geometry coder can be easily integrated with a class of the best existing connectivity coders for tetrahedral meshes [GGS99, YMC00], which are lossless, so that we can compress both geometry and connectivity losslessly, for both steady-state and time-varying datasets. We remark that some extra cost is associated with this integration, as discussed at the end of this section. However, our compression results are still superior after offsetting such extra cost, as will be seen in Section 4.

The technique of [GGS99] achieves an average bit rate for connectivity of about 2 bits per tetrahedron, which is still the best reported result so far; the technique of [YMC00] simplifies that of [GGS99], with a slightly higher bit rate. Both techniques are based on the same traversal and encoding strategy, reviewed as follows. Starting from some tetrahedron, adjacent tetrahedra are traversed. A queue, \( Q \), called partially visited triangle list, is used to maintain the triangles whose tetrahedron on one side of the triangle has been visited but the one on the other side has not. When going from the current tetrahedron to the next tetrahedron sharing a common triangle \( t \), the new tetrahedron is encoded by specifying its fourth vertex, say \( v \). If \( v \) is contained in some triangles in \( Q \), \( v \) is encoded by a small local index into an enumeration of a small number of candidate triangles in \( Q \) that are (necessarily) adjacent to the current triangle \( t \). If \( v \) cannot be found from these candidate triangles, an attempt is tried to encode \( v \) by a global index into the set of all vertices visited so far (namely, using \( \log_2 k \) bits if \( k \) vertices have been visited so far). Finally, if \( v \) has never been visited before, then \( v \) is recorded by its full geometry coordinates.

Now we describe how to integrate our geometry coder with the above scheme. We perform our geometry compression first. As described in Section 3.1, at the end of Step 3, after clustering and re-ordering the vertex list, we update the cell list so that its vertex indices are the new indices of the corresponding vertices in the new vertex list. Specifically, after the vertex re-ordering, we produce a list of tuples \([VID_{old}, VID_{new}]\), meaning that the vertex with index \( VID_{old} \) in the original vertex list now has index \( VID_{new} \) in the new vertex list. We then go over the cell list and update the vertex indices accordingly. Note that we now still maintain the original connectivity information with the cell list, and can discard the list of tuples \([VID_{old}, VID_{new}]\). After our geometry compression is complete, the above connectivity compression scheme can be performed in exactly the same way, except that for the last case, when \( v \) has never been visited before, we use the (new) index to the (new) global vertex list for \( v \) (using \( \log_2 n \) bits for \( n \) vertices), rather than the full geometry information of \( v \). In this way, the connectivity compression operates in the same way, with the “base geometry data” being the indices to the vertex list, rather than the direct geometry information of the vertices. To decompress, we first decode our geometry code, and then the connectivity code. Given the vertex indices, the corresponding actual geometry information is obtained from our decoded vertex list. It is easy to see that this integration scheme works for time-varying datasets as well.

The above integration scheme causes an extra cost of \( \log_2 n \) bits per vertex, because the vertex list, after re-ordering by our geometry coder, is fixed and may not be in the same order as that in which the vertices are first visited in the connectivity-coding traversal. Therefore, during such traversal, when we visit a vertex \( v \) for the first time, we use the index to the global vertex list to represent \( v \), resulting in a cost of \( \log_2 n \) bits per vertex for encoding such “vertex permutation”. Note that other geometry coders based on connectivity traversal such as Flipping (see Section 4) do not have this issue, as the vertex list is reconstructed in the same order of the first visit of the vertices in that traversal.

Our next task is to reduce such extra cost by encoding the permutation sequence (the sequence of vertex indices produced by connectivity traversal). Our idea is that the connectivity traversal goes through local vertices, which should also be near neighbors in our TSP order; in other words, the two sequences should be somehow correlated, which we can explore. Our solution is a simple one: we perform a differential coding on the permutation sequence. Typically there are many distinct index differences; we again encode them by the two-layer modified Huffman code using the greedy heuristic for alphabet partitioning (see Section 3.4). We show in Section 4 that this approach encodes the permutation sequence quite efficiently.

4. Results

We have implemented our lossless geometry compression techniques in C++/C, and run our experiments on a Sun Blade 1000 workstation with dual 750MHz UltraSPARC III CPUs and 4GB of main memory. Although tetrahedral volume datasets are widely used in scientific applications, most of them are classified and are not publicly available; we thus resorted to the use of the ones tetrahedralized from curvilinear datasets for most of our test data (except for one dataset that was originally a tetrahedral mesh\(^1\)). Note that such meshes still have their vertices irregularly sampled over the volumes, and hence serve the same purpose as real tetrahedral meshes in testing the effectiveness of our geometry compression techniques.

\(^1\) All the datasets used were already tetrahedral meshes in the format of vertex and cell lists described at the beginning of Section 3 when we acquired them, which have been benchmark datasets of tetrahedral meshes in the visualization literature.
The datasets we tested are listed in Table 1. The Spx dataset is a tetrahedral mesh, and the remaining datasets have been tetrahedralized from curvilinear datasets, where Tpost10 and Tpost20 are of the same mesh with 10 and 20 time steps respectively. These are all well-known datasets obtained from real-world scientific applications: the Spx dataset is from Lawrence Livermore National Lab, the Blunt Fin (Blunt), the Liquid Oxygen Post (Post), the Delta Wing (Delta), and the Tpost datasets are courtesy of NASA. The Combustion Chamber (Comb) dataset is from Vtk [SML96]. We show their representative isosurfaces in Figure 1.

Re-ordering Vertex List
To evaluate the effectiveness of our approach of re-ordering vertex list, we show in Table 2 the resulting entropy after performing the re-ordering on a representative dataset. Specifically, we compare the following re-ordering approaches: (1) no re-ordering, i.e., using the original order in the input vertex list; (2) sorting, i.e., partitioning the vertices into various numbers of clusters, which involves sorting the vertices; (3) TSP with simulated annealing (TSP-ann), i.e., first partitioning the vertices into clusters, and then re-ordering each cluster by solving the TSP problem using simulated annealing; and (4) TSP with minimum spanning tree (TSP-MST), i.e., the same as (3) but solving the TSP problem with the minimum-spanning-tree heuristic. After re-ordering, we replace each vertex component by its difference from the corresponding component of the previous vertex, and compute the 8-bit entropy as follows: the 32-bit differences are treated as four bytes, and a separate entropy is computed for the most significant byte, the second most significant byte, and so on. We then sum the separate entropy of each 8 bits. Table 2 also shows the corresponding re-ordering times.

It is easy to see from Table 2 that re-ordering clearly reduces the entropy, with TSP-ann giving the best entropy, followed by TSP-MST, and then sorting. The running times are in reverse order, showing a nice trade-off between quality and speed. Observe that TSP-MST produces entropies comparable to those of TSP-ann, but the speed can be more than 11 times faster. Also, it is interesting to see that as we partition into more clusters, the running times of TSP-MST and TSP-ann both reduce significantly, with similar entropy values (surprisingly, the entropy become smaller in some cases, but typically the compression ratios become slightly worse, as will be seen later). We found that a partition in which there were about 100–200 vertices per cluster typically gave the best performance—very fast with equally competitive compression. In summary, TSP-MST with a cluster size of 100–200 vertices (e.g., about 512 clusters for our datasets) is a right choice for both good compression and fast computing.

Encoding for Steady-State Data
We compare the following encoding algorithms on our test data (CGreedy is our proposed method and others are to be compared against):

1. **Dynamic Programming:** We partition and cluster vertices and then re-order them within each cluster by solving TSP with either a simulated annealing (TSP-ann) or a minimum-spanning-tree heuristic (TSP-MST). Then for each cluster we encode each vertex-component difference by a separate, optimal modified Huffman code using the dynamic programming method of [CCMW03]. For steady-state data, there are four such Huffman codes constructed for each cluster, and a total of $4K$ Huffman codes for $K$ clusters. The cost of storing the Huffman tables is included in the compression results.

2. **Greedy:** Same as above but the modified Huffman code is constructed using the greedy heuristic given in [CCMW03].
3. Combined Greedy (CGreedy): Same as Greedy but we try to reduce the overhead of keeping 4K Huffman tables for K clusters: we only keep two Huffman tables in total, one for all coordinates across all clusters, and the other for all scalar values across all clusters. We describe the one for the coordinates; the other is similar. For each cluster, we concatenate the x-sequence (first the x-value of the first vertex, followed by the x-value differences of the remaining vertices), the y-sequence (analogous to the x-sequence), and the z-sequence together into a single cluster sequence. We concatenate all such cluster sequences into one global sequence by visiting clusters in a fixed order, and construct a two-layer modified Huffman code for this global sequence (and try to derive an optimal alphabet partitioning using the same greedy approach). All the encoding steps are the same as the greedy method, except that we are no longer restricted to a single x-, y-, z-, or scalar-value sequence within a local cluster.

4. 8-bit Modified Huffman (8-bit MH): Same as Greedy but the encoding is done by a fixed modified Huffman code instead. That is, we encode the three most significant bytes (MSB) of each symbol with a Huffman code and then code the remaining 8 bits as an offset.

5. Gzip: Same as Greedy but the encoding is done by gzip, the well known open source LZ77 [ZL77] based string compression technique.

6. Flipping: We extend the most popular flipping approach [TG98] for triangle meshes to tetrahedral meshes: rather than predicting the current vertex position by flipping from the opposite vertex of an edge-adjacent triangle through the center of the common edge, we predict by flipping from the opposite vertex of a face-adjacent tetrahedral cell through the center of the common face. Note that this method quantizes the coordinates and scalar values as the first step. Also, it works on the entire mesh and does not partition vertices into clusters. The traversal is according to the connectivity coder of [YMC00]. We encode the prediction errors by two methods: Gzip and 8-bit Huffman coding (one Huffman code for each of the four 8-bit symbols, for each quantized 32-bit vertex component; similarly for other (8c)-bit quantizations).

In Table 3, we show the compression results of the above encoding algorithms, together with TSP-MST or TSP-ann, with no quantization of coordinates/scalar values (and hence Flipping is not compared here), for the steady-state test data. The related encoding times are given in Table 4. Comparing among CGreedy, Greedy, Dynamic, 8bit MH and Gzip, it is clear from Table 3 that CGreedy gives excellent compression ratios and is almost always the winner. We observe from Table 4 that the encoding time of CGreedy, not including the time for TSP computation, is almost linear in the input size (and is independent of the number of clusters we have). This is particularly advantageous over the \( O(n^3) \)-time dynamic programming approach. Clearly, CGreedy outperforms Dynamic in both compression ratio and running time, and should be the method of choice. Also, the compression ratios obtained by using TSP-MST are comparable with those obtained with TSP-ann; however, TSP-MST is typically about 10 times as fast (see Table 2). Obviously, using TSP-MST for re-ordering and CGreedy for encoding is the method of choice that achieves both excellent compression ratios and fast running times. We also see that when we increase the number of clusters from 216 to 512, the compression becomes slightly worse (Table 3) while the TSP-MST running time can be significantly better (Table 4), showing the flexibility of our technique. It should be pointed out that the decoding part of our approach only involves table look-up operations, which takes linear time and runs very fast. For example, for the instances in Table 4, the decoding times of CGreedy range from 1.13 to 22.68 seconds. The decoding times of Greedy and Dynamic, however, are both much faster (ranging from 0.4 to 4.62 seconds). We speculate this to be due to cache-miss effects: CGreedy results in global Huffman tables which are bigger than local Huffman tables in each cluster, thereby increasing the chance of cache misses in the random table-look-up process.

In order to compare with state-of-the-art geometry compression techniques, we first quantized each vertex component (coordinate and scalar value) into a 32-bit integer, and ran CGreedy as well as Flipping; the results are shown in Table 5. It is interesting to see that between the encoding methods of Flipping, gzip is better than 8-bit Huffman coding in three out of four instances. Also, to our surprise, for Spx only 2896 vertices (14.402%) are referenced by the cells and thus the remaining vertices cannot be predicted by Flipping. Therefore we do not list the results for Spx. We show the results of CGreedy after adding the raw cost (“+ log\_2 n”) and the encoding cost (“+ perm”) of vertex permutation when in-

<table>
<thead>
<tr>
<th>Time (sec)</th>
<th>CGreedy</th>
<th>Greedy</th>
<th>Dyn</th>
<th>TSP-MST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spx 216</td>
<td>2.43</td>
<td>12</td>
<td>65</td>
<td>33</td>
</tr>
<tr>
<td>Spx 512</td>
<td>2.46</td>
<td>28</td>
<td>59</td>
<td>33</td>
</tr>
<tr>
<td>Blunt 216</td>
<td>12.66</td>
<td>19</td>
<td>92</td>
<td>86</td>
</tr>
<tr>
<td>Blunt 512</td>
<td>12.63</td>
<td>34</td>
<td>61</td>
<td>52</td>
</tr>
<tr>
<td>Comb 216</td>
<td>9.99</td>
<td>14</td>
<td>691</td>
<td>102</td>
</tr>
<tr>
<td>Comb 512</td>
<td>10.31</td>
<td>42</td>
<td>165</td>
<td>56</td>
</tr>
<tr>
<td>Post 216</td>
<td>45</td>
<td>19</td>
<td>1764</td>
<td>239</td>
</tr>
<tr>
<td>Post 512</td>
<td>55</td>
<td>44</td>
<td>371</td>
<td>56</td>
</tr>
<tr>
<td>Delta 216</td>
<td>335</td>
<td>25</td>
<td>35219</td>
<td>12318</td>
</tr>
<tr>
<td>Delta 512</td>
<td>375</td>
<td>43</td>
<td>6599</td>
<td>564</td>
</tr>
</tbody>
</table>

Table 4: Total encoding times of our compression methods, for 216 and 512 clusters, after applying TSP-MST for vertex-list re-ordering. The running times of TSP-MST are not included and are listed separately.
Table 3: Compression results with no quantization, for 216 and 512 clusters. The bit rate before compression is 128 b/v.

<table>
<thead>
<tr>
<th>Size (b/v)</th>
<th>TSP-MST</th>
<th>TSP-ann</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spx 216</td>
<td>102.11</td>
<td>8bit MH</td>
</tr>
<tr>
<td>Spx 512</td>
<td>105.41</td>
<td>8bit MH</td>
</tr>
<tr>
<td>Blunt 216</td>
<td>44.24</td>
<td>8bit MH</td>
</tr>
<tr>
<td>Blunt 512</td>
<td>47.82</td>
<td>8bit MH</td>
</tr>
<tr>
<td>Comb 216</td>
<td>87.53</td>
<td>8bit MH</td>
</tr>
<tr>
<td>Comb 512</td>
<td>89.44</td>
<td>8bit MH</td>
</tr>
<tr>
<td>Post 216</td>
<td>43.51</td>
<td>8bit MH</td>
</tr>
<tr>
<td>Post 512</td>
<td>46.45</td>
<td>8bit MH</td>
</tr>
<tr>
<td>Delta 216</td>
<td>74.94</td>
<td>8bit MH</td>
</tr>
<tr>
<td>Delta 512</td>
<td>75.68</td>
<td>8bit MH</td>
</tr>
</tbody>
</table>

Table 5: Compression results with 32-bit quantization (128 b/v before compression). CGreedy uses TSP-MST for vertex re-ordering. “Org.” lists the original compression results; “+ log2 n” adds log2 n b/v to “Org.”, and “+ perm” adds the encoding cost of vertex permutation to “Org.”, which shows our final results (in bold). Flipping does not partition vertices into clusters.

<table>
<thead>
<tr>
<th>Size (b/v)</th>
<th>Org.</th>
<th>CGreedy</th>
<th>Flipping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blunt 216</td>
<td>25.00</td>
<td>40.32</td>
<td>31.44</td>
</tr>
<tr>
<td>Blunt 512</td>
<td>26.04</td>
<td>41.36</td>
<td>32.40</td>
</tr>
<tr>
<td>Comb 216</td>
<td>85.08</td>
<td>100.60</td>
<td>94.52</td>
</tr>
<tr>
<td>Comb 512</td>
<td>86.89</td>
<td>102.41</td>
<td>97.00</td>
</tr>
<tr>
<td>Post 216</td>
<td>24.69</td>
<td>41.43</td>
<td>29.62</td>
</tr>
<tr>
<td>Post 512</td>
<td>25.74</td>
<td>42.48</td>
<td>30.66</td>
</tr>
<tr>
<td>Delta 216</td>
<td>60.29</td>
<td>77.98</td>
<td>71.74</td>
</tr>
<tr>
<td>Delta 512</td>
<td>60.39</td>
<td>78.08</td>
<td>68.87</td>
</tr>
</tbody>
</table>

now, we perform yet another set of compression experiments on Tpost10, trying to improve the compression ratio as much as we can. We observe that the scalar values across different time steps may not differ much for each vertex, and hence we want to do differential coding also along the direction of the time steps. For each vertex entry, we first take differences between scalar values of adjacent time steps. For the resulting vertex entries, we define the distance between two vertices to be the sum of the log of the component difference between the two vertex entries, over all components. We then perform TSP computation according to this new distance definition. Each CGreedy is similar to its previous definition: the x-, y-, z- items are all combined into one Huffman code, and the scalar values (differences) are combined into one Huffman code for every i time steps as a unit. The results are presented in Table 7. As shown, the TSP computa-
5. Conclusions

We have presented a novel lossless geometry compression technique for steady-state and time-varying irregular grids represented as tetrahedral meshes. Our technique exhibits a nice trade-off between compression ratio and encoding speed. Among the options provided in various stages, the following version of our approach gives the best choice in terms of both good compression and fast computing: a partition with cluster size of about 100–200 vertices, TSP-MST for vertex-list re-ordering, and CGreedy and CGreedyi respectively for steady-state and time-varying vertex-difference encoding. Our technique achieves superior compression ratios with reasonable encoding times, with very fast (linear) decoding times. We also show how to integrate our geometry coder with the state-of-the-art connectivity coders, and how to reduce the integration overhead by compressing the permutation sequence. Compared with the state-of-the-art Flipping approach, our technique achieves significant improvements of up to 46.70 b/v (61.2%) for steady-state data, and up to 83.56 b/v (28.9%) for time-varying data.

One novel feature of our geometry coder is that it does not need any connectivity information. This makes it readily applicable to the compression of point cloud data, which is becoming increasingly important recently. Our on-going work is to pursue this research direction. Another important feature is that our technique can be used with both low precision quantized data or with high precision unquantized data, making it suitable for a wide range of applications, including those scientific ones for which quantization is not acceptable.

Although we have focused on compression for archiving and transmission, we can extend and integrate our approach into the data visualization process. Given the large dataset sizes, it is essential to integrate compression techniques into out-of-core visualization techniques which guarantee that at any time no more data than can fit in main memory is needed, to ensure fast visualization. We believe that our clustering technique already facilitates such a possibility, and our future research shall focus on integrating our compression technique with the visualization process.

Table 6: Compression results (with no quantization) and encoding times on Tpost10. We fix the number of clusters to 512. The bit rate before compression is 416 b/v. The TSP is performed with the distance between two vertices $v_1 = (x_1, y_1, z_1, f_{11}, f_{12}, ...) \text{ and } v_2 = (x_2, y_2, z_2, f_{21}, f_{22}, ...)$ defined as $\log |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|$. The TSP computation is done by TSP-MST, and takes 339 seconds.

<table>
<thead>
<tr>
<th>Method</th>
<th>CGreedy1</th>
<th>CGreedy5</th>
<th>CGreedy10</th>
<th>Greedy</th>
<th>Gzip</th>
<th>8bit MH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (b/v)</td>
<td>172.57</td>
<td>163.34</td>
<td>160.64</td>
<td>209.34</td>
<td>214.17</td>
<td>337.52</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>153</td>
<td>335</td>
<td>500</td>
<td>36</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Additional compression results (b/v) and encoding times (sec) on Tpost10 with 512 clusters and no quantization. The bit rate before compression is 416 b/v. A differential coding along the time steps is used, and the TSP computation uses a new definition of distance. The TSP computation uses TSP-MST, and takes 431 seconds.

<table>
<thead>
<tr>
<th>Method</th>
<th>CGreedy1</th>
<th>CGreedy5</th>
<th>CGreedy10</th>
<th>Greedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>166.14</td>
<td>159.40</td>
<td>157.71</td>
<td>206.87</td>
</tr>
<tr>
<td>Time</td>
<td>173</td>
<td>299</td>
<td>433</td>
<td>36</td>
</tr>
</tbody>
</table>

References


Table 8: Compression results with 24-bit quantization. Before compression, Tpost10 is 312 b/v and Tpost20 is 552 b/v. Our approaches (CGreedyi) are the same as in Table 7. We also list the results of adding the extra cost of encoding the permutation sequence (“+perm”), which are our final results (in bold). Flipping does not partition vertices into clusters.

![Figure 1: Representative isosurfaces from our test datasets. Top row: left—Spx; middle—Blunt; right—Comb. Bottom row: left—Post; middle—Delta; right—Tpost10 (Tpost20).](image)


submitted to EUROGRAPHICS 2005.