Eurographics/ IEEE-VGTC Symposium on Visualization (2006)
Thomas Ertl, Ken Joy, and Beatriz Santos (Editors)

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

Dan Chen and Yi-Jen Chiang and Nasir Memon and Xiaolin Wu

Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11201, USA

Abstract

In this paper we investigate the problem of lossless geometry compression of irregular-grid volume data represented as a tetrahedral mesh. We propose a novel lossless compression technique that effectively predicts, models, and encodes geometry data for both steady-state (i.e., with only a single time step) and time-varying datasets. Our geometry coder is truly lossless and also does not *need any connectivity information. Moreover, it can be easily integrated with a class of the best existing connectivity compression techniques for tetrahedral meshes with a small amount of overhead information. We present experimental results which show that our technique achieves superior compression ratios, with reasonable encoding times and fast (linear) decoding times.*

1. Introduction

Although there has been a significant amount of research done on tetrahedral mesh compression, most techniques reported in the literature have mainly focused on compressing *connectivity* information, rather than *geometry* information which consists of *vertex-coordinates* and *data attributes* (such as *scalar values* in our case). As a result, while connectivity compression 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 geometry compression 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 typically about 4.5 times the number of vertices and that connectivity compression results in about 2 bits per cell, it is clear that geometry compression is the bottleneck in graphics compression. The situation gets worse for time-varying datasets where each vertex can have hundreds or even thousands of time-step scalar values.

The disparity between bit rates needed for representing connectivity and the geometry gets further amplified when *lossless* compression is required. Interestingly, whereas almost all connectivity compression techniques are lossless, geometry compression results in the literature almost always include a quantization step which makes them *lossy* (the only exception is the recent technique of [ILS04] for polygonal meshes). While lossy compression might be acceptable for usual graphics models to produce an approximate visual effect to "fool the eyes," it is often not acceptable for scientific applications where lossless compression is desired. Moreover, scientists usually do not like their data to be altered in a process outside their control, and hence often avoid using any lossy compression [ILS04].

In this paper we develop a truly lossless geometry compression techniques for tetrahedral volume data, for both steady-state and time-varying datasets. By truly lossless we mean a technique where quantization of vertex coordinates is not required. However, the technique works effectively even if quantization is performed. We take a novel direction in that our geometry coder is *independent* of connectivity coder (albeit they can be easily incorporated) and re-orders the vertex list *differently* from connectivity-coder traversal. This incurs an additional overhead for recording the permutation of the 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

[†] Research supported by NSF Grant CCF-0118915; research of the second author is also supported by NSF CAREER Grant CCF-0093373. Author E-mail: dchen@cis.poly.edu; {yjc, memon,xwu}@poly.edu.

© The Eurographics Association 2006.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

not "burdened" by connectivity compression. Secondly, such a 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. It turns out that our geometry coder significantly improves over the state-of-the-art techniques even after paying the extra cost for representing vertex ordering. Moreover, we show that the vertex ordering can be efficiently encoded, resulting in even greater improvements in overall compression rates.

Our method makes use of several key ideas. We formulate the problem of optimally re-ordering the vertex list as a combinatorial optimization problem, namely, the *traveling salesperson problem* (TSP), and solve it with heuristic algorithms. To obtain 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 arithmetic/Huffman code* based on an *optimized alphabet partitioning* approach using a greedy heuristic. Our geometry compression can also be easily integrated with the best *connectivity* compression techniques for tetrahedral meshes [GGS99, YMC00] with a small amount of overhead.

Experiments show that our technique achieves superior compression ratios, with reasonable encoding times and fast (linear) decoding times. To the best of our knowledge the results reported in this paper represent the best that has been achieved for lossless compression of geometry data for tetrahedral meshes both in terms of entropy of prediction error as well as final coded bit-rate. 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 achieves significant improvements of up to 62.09 bits/vertex (b/v) (67.2%) for steady-state data, and up to 61.35 b/v (23.6%) for time-varying data.

2. Previous Related Work

There are relatively few results that focus on geometry compression. 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. Also, their techniques are for *multi-resolution* compression, rather than for *single-resolution* compression as we consider in this paper. (We refer to [GD02, Hop98, KSS00] and references therein for other multi-resolution methods.)

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, CCMW05b]. 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.

3. Our Approach

In this section we develop the main ideas behind our technique. In the following, we assume that each vertex v_i in the vertex list consists of a $t+3$ tuple $(x_i, y_i, z_i, f_{i1}, f_{i2}, \cdots, f_{it})$ where x_i, y_i, z_i are the coordinates and $f_{i1}, \cdots, f_{it}$ are the scalar values for t time steps, with steady-state data ($t = 1$) being just a special case. From now on, 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 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 capture the underlying structure in the data with a simple-to-describe *prediction model*. Then, we 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 is by its difference with respect to some neighboring vertex. This is called *differential coding*.

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 orders will lead to different compression performance with differential coding. This is the main idea behind our compression technique. The idea of re-ordering to benefit compression has been explored before for lossless image compression [MSM95]. However, its application to geometry coding is novel and as we show later, quite effective. The reason why it works is that any re-ordering technique typically involves the need for overhead information to specify the order. However, in the case of geometry compression, this overhead is a small fraction of the total cost and turns out to be well worth the effort.

We formulate the problem of how to re-order the vertex list to get maximum compression as a combinatorial optimization problem, i.e., the *traveling salesperson problem* (TSP), which is NP-hard, 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 reducing this overhead in Section 3.5.

For the second task of encoding the prediction errors/residuals, we observe that the 8-bit exponents only have relatively few distinct values and together with the sign bit (we call the 9-bit concatenation a "signed exponent" for convenience) they can be effectively compressed by a simple gzip. Our focus is thus on the mantissa. For this reason, the vertex partitioning and TSP re-ordering steps mentioned above will be solely based on the mantissa values (see below). To encode the mantissa of the prediction errors, a simple approach would be to use an entropy coding such as an arithmetic or Huffman code. The problem is that such mantissa difference values are taken from a very large alphabet (23-bit numbers with many distinct values) and the corresponding Huffman table or probability table for static and semi-adaptive arithmetic coding would be so large that it would offset any gain made by the coding technique itself. The problem is more than just 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. To address this problem the common approach applied in the literature has been to arithmetic/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 arithmetic/Huffman code* to solve this problem.

In summary, our technique for lossless geometry compression consists of the following stages:

1. **Step 1. Take differences along time steps.** This step is done for time-varying data only. For each vertex, the mantissa of the scalar value at time step i is replaced by its difference with the mantissa of the scalar value at time step $i-1$.
2. **Step 2. Partition vertices into clusters.** This is described in more detail below in Section 3.3.
3. **Step 3. Re-order the vertices by formulating and solving a TSP problem.** This is described in more detail below in Section 3.2.
4. **Step 4. Take mantissa differences.** The mantissa of each element in vertex v_i is replaced by its difference with the mantissa of the corresponding element in the previous vertex v_{i-1}.
5. **Step 5. Entropy code the mantissa differences.** We gather frequencies of each element in the mantissa difference tuples and construct a two-layer modified arithmetic or Huffman code.
6. **Step 6. Compress the signed exponents.** Finally, we compress the signed exponents for all x-values, for all y-values, etc., and for all scalar values at time step i, $i = 1, 2, \cdots$, in that order, using gzip.

We remark that our approach can be easily modified if an initial quantization step (and thus a *lossy* compression) is desired: the quantized integers play the roles of the mantissa values, and we do not need to worry about the exponents. As for the above lossless algorithm, Steps 1, 4 and 6 are trivial and do not need further details. We elaborate the other steps in Sections 3.2–3.4. In Section 3.5 we describe how to integrate our geometry coder with a class of the best existing connectivity compression methods [GGS99, YMC00].

3.2. Step 3: Vertex List Re-ordering

The efficiency of differential coding of the mantissa values of the vertex list depends on the the manner this list is ordered. Given an unordered list of vertices $\{v_1, v_2, \ldots, v_n\}$, we want to compute a permutation π such that the objective function $\sum_{i=2}^{n} C(|v_{\pi(i)} - v_{\pi(i-1)}|)$ is minimized. Here, the function $C(\cdot)$ represents the cost in bits of representing the mantissa difference of two adjacent vertices. For arriving at a cost function we make another simplifying assumption that representing the value n is simply proportional to $\log_2 n$. This allows us to restate the problem 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 mantissa 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 $v_i = (x_i, y_i, z_i, f_{i1}, f_{i2}, \cdots, f_{it})$ and $v_j = (x_j, y_j, z_j, f_{j1}, f_{j2}, \cdots, f_{jt})$ to be $\lg|\bar{x}_i - \bar{x}_j| + \lg|\bar{y}_i - \bar{y}_j| + \lg|\bar{z}_i - \bar{z}_j| + \lg|\bar{f}_{i1} - \bar{f}_{j1}| + \cdots + \lg|\bar{f}_{it} - \bar{f}_{jt}|$[†], where $\bar{a}$ means the mantissa value of a.

Unfortunately TSP is in general an NP-complete problem, but 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

[†] We treat the case when the argument is zero as special and simply return a zero for the log value.

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 no more than twice the optimal path length if the distances in G obey the 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.

Finally, 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 2: 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. 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 mantissa of 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 mantissa of the y-values and again split them equally into $K^{1/3}$ groups. Finally we repeat the process by the mantissa of the z-values. Each resulting group is a cluster of the same size, and the vertices in the same group are spatially close (in terms of the mantissa values of the coordinates). 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 (due to sorting).

3.4. Step 5: Entropy Coding Mantissa Differences

As mentioned in Section 3.1, vertex mantissa differences have a very high dynamic range and cannot be efficiently encoded in a straightforward manner. 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 [CCMW03], we formulated the problem of optimal alphabet partitioning for the design of a two-layer modified Huffman code, and gave both an $O(N^3)$-time dynamic programming algorithm and an $O(N\log N)$-time greedy heuristic, where N is the number of distinct symbols in the source sequence. The greedy method gives slightly worse (and yet comparable) compression ratios, but runs much faster.

In this paper, to encode the vertex mantissa differences, we use the above greedy algorithm to construct a two-layer modified Huffman code, and also extend the greedy algorithm in a similar way to construct a *two-layer modified arithmetic code* where in the first layer we replace the Huffman code by a semi-adaptive arithmetic code. As is well known, the latter (arithmetic code) gives better compression ratios, at the cost of slower encoding and decoding times.

Since we encode multiple clusters and multiple vertex components, we can either use a single arithmetic/Huffman code for the mantissa of each coordinate value (and scalar value) for each cluster, or we can use a single arithmetic/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 mantissa differences for the coordinate values had similar distributions and using a single arithmetic/Huffman code for them gave the best performance, since the overall probability- or Huffman-table size can be greatly reduced. The same was the case for the scalar values. Hence we only use two arithmetic/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 arithmetic/Huffman code for every i time steps across all clusters; this is the **CGreedy*i*** approach. In summary, our proposed coding methods are CGreedy for steady-state data, and CGreedy*i* for time-varying data, using two-layer modified arithmetic/Huffman coding.

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. 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.

To integrate our geometry coder with the above scheme, 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. 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 at most $\log_2 n$ 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 a "vertex permutation". 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.

Data	# cells	# vertices	vertex-list size (byte)
Spx	12936	20108	321,732
Blunt	187395	40960	655,364
Comb	215040	47025	752,404
Post	513375	109744	1,755,908
Delta	1005675	211680	3,386,884
Tpost10	615195	131072	6,815,752
Tpost20	615195	131072	12,058,632

Table 1: *Statistics of our test datasets. The entries in "vertex-list size" show the uncompressed file sizes of the vertex lists, including all coordinates and scalar values, each a binary 32-bit floating-point number.*

Entropy (b/v)	TSP-ann	TSP-MST	Sorting	Org.
Comb 125	57.38	61.54	64.53	75.71
Comb 216	58.81	61.71	64.59	
Comb 343	60.03	62.06	64.67	
Comb 512	60.40	62.46	64.79	

Table 2: *Re-ordering results in terms of 8-bit entropy (b/v), with different numbers of clusters shown in the first column. "Org." means no re-ordering.*

4. Results

We have implemented our techniques in C++/C and run our experiments. The datasets we tested are listed in Table 1[‡]. They are all given as tetrahedral meshes, where Tpost10 and Tpost20 are of the same mesh with 10 and 20 time steps respectively, and the remaining datasets are steady-state data. 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 from 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: using the original order in the input; (2) sorting: partitioning the vertices into clusters, which involves sorting the vertices; (3) TSP-ann: first partitioning the vertices into clusters, and then re-ordering each cluster by solving the TSP problem using simulated annealing; and (4)

‡ They are all available at http://cis.poly.edu/chiang/datasets.html.

TSP-MST: the same as (3) but solving the TSP problem with the minimum-spanning-tree heuristic. After re-ordering, we replace the mantissa of each vertex component by its difference from the corresponding mantissa of the previous vertex, and compute the *8-bit entropy* of the mantissa differences as follows: for a particular coordinate (or scalar value), a separate entropy is computed for the first byte, the second byte, and so on; we then sum the separate entropy of each byte.

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. We found that the running times were 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 we found that the speed could be more than 200 times faster. Also, as we partitioned into more clusters, the running times of TSP-MST and TSP-ann both reduced significantly, with similar entropy values. 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

Recall from Section 3.4 that our encoding techniques are CGreedy using two-layer modified arithmetic or Huffman coding (A-Cg or H-Cg) for the mantissa values, and gzip for the signed exponents. We show our compression results with *no quantization* (i.e., lossless compression) in Table 3. We also compare our results with those of Gzip and 8-bit adaptive arithmetic coding (AC) on the original input data. For gzip, we compressed for all x-values together, then all y-values together, and so on; this gave the best results among other gzip options. In 8-bit arithmetic coding, for each vertex component, we code the first byte and then the next byte and so on. We tried both static and adaptive arithmetic codes; in Table 3 we only list the results of the adaptive one since they were better.

As can be seen from Table 3, our results are always much better than AC, and even better than Gzip except for Blunt and Post. However, Gzip compressed amazingly well for Blunt and Post. With further investigation, we found that in these two datasets the z-coordinates only had relatively very few distinct values, i.e., the vertices lie on these relatively few z-planes, and Gzip was very good in capturing such repetitions. For this, we slightly modify our algorithm and give the z-values a special treatment: all the partitioning/clustering and the TSP-MST vertex re-ordering steps are the same; only at the final encoding step, we code the entire z-values by gzip, while the x, y and scalar values are encoded as before (A-Cg for the mantissa's and gzip for the signed exponents). We show the results in Table 4. Observe that gzip compressed the z-values from 32 b/v to 0.07 and 0.05 b/v! With our special treatment for z, our results are now better than Gzip for these two special datasets (see Table 4). In general, we can first try to gzip each of the entire x, y, z and scalar values to see if any of them deserves a special treatment.

	TSP-MST			Original	
Size(bpv)	H-Cg	A-Cg	Exp	AC	Gzip
Spx 216	91.4	89.2	14.1	107.2	112.7
Spx 512	90.7	88.1	13.9		
Blunt 216	41.6	40.6	7.8	108.6	25.9
Blunt 512	41.5	40.3	7.6		
Comb 216	68.2	67.2	5.7	98.5	100.7
Comb 512	68.9	68.1	5.9		
Post 216	36.6	35.8	8.0	105.3	28.1
Post 512	37.8	37.2	8.8		
Delta 216	74.7	70.1	11.6	100.9	126.4
Delta 512	74.0	70.3	11.0		

Table 3: *Compression results (b/v) with* no *quantization, for 216 and 512 clusters. The bit rate before compression is 128 b/v. The cost of signed exponents is included in H-Cg and A-Cg; we also list this exponent cost separately (36 b/v before compression). We compare our results with Gzip and 8-bit adaptive arithmetic coding on the original input data.*

Size(bpv)	Exp	A-Cg	Z	Total	Gzip
Blunt 216	2.79	19.89	0.07	22.75	25.85
Blunt 512	2.67	19.62	0.07	22.37	25.85
Post 216	4.94	19.79	0.05	24.78	28.09
Post 512	5.27	19.94	0.05	25.26	28.09

Table 4: *Compression results (b/v) with* no *quantization and the special treatment of the z coordinate.*

In addition to Gzip and AC, it is natural to compare with Flipping, an easy extension of the most popular flipping algorithm [TG98] from triangle meshes to tetrahedral meshes. Flipping is widely considered the state of the art when applied *after quantization*, and floating-point flipping methods (with *no* quantization) were recently given in [ILS04] for polygonal meshes. We computed the 8-bit entropy of the prediction errors of lossless Flipping, as well as the 8-bit entropy of the original input data, as shown in Table 5[§]. Interestingly, lossless Flipping actually *increases* the entropy for *all* our datasets, including steady-state and time-varying ones (such events have been observed in [ILS04] for polygonal meshes, but only for very few datasets). Therefore, lossless Flipping is not a competitive predictor for our volume meshes and we do not compare our methods with it.

Now, we want to see the compression performance of our

§ Surprisingly, Spx has only 2896 vertices (14.402%) that are referenced by the cells and thus the remaining vertices cannot be predicted by Flipping. Hence we do not list the results for Spx.

Entropy(bpv)	Flip	Original
Blunt	100.15	90.60
Comb	104.17	97.44
Post	108.22	102.99
Delta	103.03	97.33
Tpost10	299.87	257.44
Tpost 20	538.25	447.29

Table 5: *The 8-bit entropy (b/v) of the prediction errors of* lossless *Flipping and the original input data.*

methods when an initial quantization (i.e., *lossy* compression) is desired. To this end, we first quantized each vertex component (coordinate and scalar value) into a 32-bit integer¶, and then applied our algorithm as well as the state-of-the-art Flipping approach‖. Note that now our method does not use gzip at all. To compare the prediction performance, we observed that the resulting entropies of our method were much better than those of Flipping even after adding the vertex permutation costs to our entropies when integrated with the connectivity coder (we omit the detailed numbers due to lack of space). In Table 6, we compare the results after the final encoding, where for Flipping we encoded the flipping errors by gzip and 8-bit static arithmetic coding (which was always better than 8-bit adaptive arithmetic coding at this time). To make the comparison fair we also show our results after adding the *raw cost* ("$+\lg n$") and the *encoding cost* ("+perm") of vertex permutation when integrated with the connectivity coder [YMC00]. It can be seen that we encoded the permutation sequence quite efficiently and our final results ("+perm") achieve significant improvements over Flipping (up to 62.09 b/v (67.2%)), showing the efficacy of our technique despite the additional overhead cost of encoding the permutation sequence.

Encoding for Time-Varying Data

In Table 7 we show the results of applying our compression techniques to the Tpost10 dataset, with *no* quantization. We found that as we increased *i* in CGreedy*i*, the compression ratio increased while the encoding speed decreased. Also, arithmetic coding (A-Cg*i*) compressed better than Huffman coding (H-Cg*i*) with longer encoding times, as expected. We also compared with Gzip, 8-bit adaptive and static arithmetic coding (AC(A) and AC(S)), on the original input data. (As seen in Table 5 lossless Flipping did not predict well and hence we did not compare with lossless Flipping.) We can see that Gzip is the best among the three, and our results are always significantly better than Gzip, with the best one (A-Cg10) 66.44 b/v (32.56%) more efficient.

¶ We also tried 24-bit quantization, but Blunt, Post and Delta all resulted in some vertices collapsed with inconsistent scalar values. Hence we performed 32-bit quantization for the steady-state data.

‖ In [CCMW05a] we improved the geometry compression over Flipping; however, recently we found that the connectivity-compression overhead in [CCMW05a] seemed to offset the gains in geometry compression (we are still trying to reduce this overhead but the status is not finalized yet), and thus here we did not compare with [CCMW05a].

	TSP-MST			Flipping	
Size(bpv)	A-Cg	+lg n	+perm	Gzip	AC
Blunt 216	22.36	37.69	30.13	69.7	97.2
Blunt 512	21.66	36.98	29.27		
Comb 216	78.66	94.18	84.32	105.7	97.5
Comb 512	78.87	94.39	84.72		
Post 216	22.30	39.05	30.32	95.1	92.4
Post 512	22.58	39.33	31.26		
Delta 216	55.33	73.02	66.90	75.3	91.3
Delta 512	55.23	72.92	66.22		

Table 6: *Compression results (b/v) with 32-bit quantization (with* no *special treatment for z). The flipping errors were encoded with gzip and 8-bit static arithmetic coding.*

	A-Cg20	A-Cg10	F-Gzip	F-AC
Tpost10	N/A	105.60	166.17	148.92
+perm	N/A	112.41		
Tpost20	192.24	195.78	289.03	260.39
+perm	199.04	202.60		

Table 8: *Compression results (b/v) with 24-bit quantization. Before compression, Tpost10 is 312 b/v and Tpost20 is 552 b/v. Our approaches used 512 clusters and TSP-MST.*

Finally, in Table 8, we compare the results of applying our method as well as Flipping after an initial 24-bit quantization was performed. For Flipping, we show the results of using gzip and 8-bit static arithmetic coding to encode the flipping errors (this time the static arithmetic coding was always better than the adaptive one, and in fact better than gzip as well). For our method, we also list the results of adding the extra cost of encoding the permutation sequence ("+perm"), which are our final results. We see that our approaches are always significantly better than Flipping, with the best improvement (A-Cg20 vs. F-AC for Tpost20 +perm) up to 61.35 b/v (23.6%).

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. 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

	TSP-MST							Original		
	H-Cg1	H-Cg5	H-Cg10	A-Cg1	A-Cg5	A-Cg10	Exp	Gzip	AC(A)	AC(S)
Size	165.97	157.58	155.40	149.70	140.00	137.63	6.414	204.07	262.09	269.54

Table 7: *Compression results (b/v) on Tpost10 with 512 clusters and* no *quantization. The cost of signed exponents (Exp) is included in H-Cgi and A-Cgi. The bit rate before compression is 416 b/v. The TSP computation used TSP-MST.*

to reduce the integration overhead by compressing the permutation sequence.

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; some preliminary results of this follow-up work are given in [CCM05].

References

[AD01] ALLIEZ P., DESBRUN M.: Valence-driven connectivity encoding for 3D meshes. *Computer Graphics Forum 20*, 3 (2001. Special Issue for Eurographics '01), 480–489.

[CCM05] CHEN D., CHIANG Y.-J., MEMON N.: Lossless compression of point-based 3D models. In *Proc. Pacific Graphics* (2005), pp. 124–126. (Short paper/poster).

[CCMW03] CHEN D., CHIANG Y.-J., MEMON N., WU X.: Optimal alphabet partitioning for semi-adaptive coding of sources with unknown sparse distributions. In *Proc. Data Compression* (2003), pp. 372–381.

[CCMW05a] CHEN D., CHIANG Y.-J., MEMON N., WU X.: Geometry compression of tetrahedral meshes using optimized prediction. In *Proc. European Conference on Signal Processing* (2005).

[CCMW05b] CHEN D., CHIANG Y.-J., MEMON N., WU X.: Optimized prediction for geometry compression of triangle meshes. In *Proc. Data Compression* (2005), pp. 83–92.

[CLRS01] CORMEN T. H., LEISERSON C. E., RIVEST R. L., STEIN C.: *Introduction to Algorithms*, 2nd ed. MIT Press, Cambridge, MA, 2001.

[DG00] DEVILLERS O., GANDOIN P.-M.: Geometric compression for interactive transmission. In *Proc. Visualization '00* (2000), pp. 319–326.

[GD02] GANDOIN P.-M., DEVILLERS O.: Progressive lossless compression of arbitrary simplicial complexes. *ACM Trans. Graphics 212*, 3 (2002), 372–379. Special Issue for SIGGRAPH '02.

[GGS99] GUMHOLD S., GUTHE S., STRASER W.: Tetrahedral mesh compression with the cut-border machine. In *Proc. Visualization '99* (1999), pp. 51–58.

[Hop98] HOPPE H.: Efficient implementation of progressive meshes. *Computers & Graphics 22*, 1 (1998), 27–36.

[IA02a] ISENBURG M., ALLIEZ P.: Compressing hexahedral volume meshes. In *Proc. Pacific Graphics* (2002).

[IA02b] ISENBURG M., ALLIEZ P.: Compressing polygon mesh geometry with parallelogram prediction. In *Proc. Visualization* (2002), pp. 141–146.

[IG03] ISENBURG M., GUMHOLD S.: Out-of-core compression for gigantic polygon meshes. *ACM Trans. Graphics 22*, 3 (2003), 935–942. Special Issue for SIGGRAPH '03.

[ILS04] ISENBURG M., LINDSTROM P., SNOEYINK J.: Lossless compression of floating-point geometry. In *Proc. CAD'3D* (2004).

[KG02] KRONROD B., GOTSMAN C.: Optimized compression for triangle mesh geometry using prediction trees. In *Proc. Sympos. on 3D Data Processing, Visualization and Transmission* (2002), pp. 602–608.

[KSS00] KHODAKOVSKY A., SCHRÖDER P., SWELDENS W.: Progressive geometry compression. In *Proc. ACM SIGGRAPH* (2000), pp. 271–278.

[LAD02] LEE H., ALLIEZ P., DESBRUN M.: Angle-analyzer: A triangle-quad mesh codec. *Computer Graphics Forum 21*, 3 (2002. Special Issue for Eurographics '02), 383–392.

[MSM95] MEMON N. D., SAYOOD K., MAGLIVERAS S. S.: Lossless image compression with a codebook of block scans. *IEEE Journal on Selected Areas of Communications 13*, 1 (January 1995), 24–31.

[Ris84] RISSANEN J. J.: Universal coding, information, prediction and estimation. *IEEE Transactions on Information Theory 30* (1984), 629–636.

[Ros99] ROSSIGNAC J.: Edgebreaker: Connectivity compression for triangle meshes. *IEEE Trans. Visualization Computer Graphics 5*, 1 (1999), 47–61.

[SML96] SCHROEDER W., MARTIN K., LORENSEN W.: *The Visualization Toolkit*. Prentice-Hall, 1996.

[TG98] TOUMA C., GOTSMAN C.: Triangle mesh compression. In *Proc. Graphics Interface* (1998), pp. 26–34.

[TR98] TAUBIN G., ROSSIGNAC J.: Geometric compression through topological surgery. *ACM Trans. Graphics 17*, 2 (1998), 84–115.

[YMC00] YANG C.-K., MITRA T., CHIUEH T.-C.: On-the-fly rendering of losslessly compressed irregular volume data. In *Proc. Visualization '00* (2000), pp. 101–108.