Grow & Fold: Compression of Tetrahedral Meshes

SM99-021

Andrzej Szymczak
School of Mathematics
Georgia Inststute of Technology
Atlanta GA 30332
andrzej@math.gatech.edu

Jarek Rossignac
Graphics, Visualization & Usability Center
College of Computing
801 Atlantic Drive
Georgia Institute of Technology
Atlanta, GA, 30332-0280
jarek@gvu.gatech.edu

October 30, 1998

1 Abstract

Standard representations of irregular finite element meshes combine vertex data (sample coordinates and node values) and connectivity (tetrahedron-vertex incidence). Connectivity specifies how the samples should be interpolated. It may be encoded for each tetrahedron as four vertex-references,

which together occupy 128 bits. Our 'Grow&Fold' format reduces the connectivity storage down to 7 bits per tetrahedron: 3 of these are used to encode the presence of children in a tetrahedron spanning tree; the other 4 constrain sequences of 'folding' operations, so that they produce the connectivity graph of the original mesh. Additional bits must be used for each handle in the mesh and for each topological 'lock' in the tree. However, as our experiments with a prototype implementation show, the increase of the storage cost due to this extra information is typically no more than 1-2%. By storing vertex data in an order defined by the tree, we avoid the need to store tetrahedron-vertex references, and facilitate variable coding techniques for the vertex data. We provide the details of simple, loss-less compression and decompression algorithms.

2 Problem Statement

This paper addresses the problem of a bit-efficient loss-less encoding of the incidence of a tetrahedral mesh whose boundary is a manifold surface. A simple representation of such a mesh consists of two tables: the *vertex table* keeping vertex coordinates and vertex data, such as temperature or pressure, and the *tetrahedron table* storing quadruples of vertex indices, representing vertex sets for each one of the m tetrahedra in the mesh (see Figure 1). The tetrahedron table describes explicitly only the vertex incidence for each tetrahedron. However, all other connectivity information, like tetrahedron-face or triangle-vertex incidence can be derived from it algorithmically. For a mesh with one Million vertices and six Million tetrahedra, the tetrahedron table requires $128m \approx 7.68 \times 10^8$ bits if 4-byte pointers are used to reference vertices or $80m \approx 4.8 \times 10^8$ bits if the vertex references are stored as 20-bit integers crossing the byte boundaries. The total size of the vertex coordinates and data (12-bit coordinates and 16-bit for a single scalar value) of such a mesh amounts to 5.2×10^7 bits: almost 10 times less. Therefore, the connectivity information dominates the storage cost and it is important that it is compressed. In this paper we are concerned only with the compression of this incidence information (described by the tetrahedron table); we do not discuss compression of the vertex data.

Our coding algorithm takes a tetrahedron table as input and produces its encoding - a string of about 7 bits per tetrahedron, thus achieving a 10-to-1 vertex

#	x	y	z	data
1				
2				
3				
$\vdots$				
n				

tetrahedron

#	v_1	v_2	v_3	v_4
1				
2				
3				
$\vdots$				
m				

Figure 1: Standard representation of a tetrahedral mesh; empirical evidence indicates that, for large meshes, m is about 4-6.7 times greater than n.

compression ratio for common meshes. The decoding algorithm is able to produce a tetrahedron table based on that string. The decoded mesh will be identical to the original one, but its tetrahedra and vertices will be listed in a different order. Even without applying any compression scheme to the vertex coordinates and data, our algorithm is able to encode both connectivity and geometry of our one Million vertex mesh using about 9.4×10^7 bits, achieving the compression ratio of about 5.7/1. Furthermore, because our scheme permits to transmit and decode the incidence independently of the vertex information, it makes it possible to use various prediction-based techniques to compress the vertex location and data ([26], [11], [12], [4]).

3 Prior Art

3.1 Representation Schemes for Tetrahedral Meshes

Numerous data structures have been proposed that combine adjacency and incidence information (see [21] for a review). Examples include winged-edge representation ([1],[2]), face-adjacency hypergraph ([7]), half-edge structure ([17],[13]), radial-edge structure ([8],[9]) and selective geometric complex ([20]). The goal behind the design of these data structures is to provide an efficient way of accessing different kinds of adjacency information without taking up much storage space. A common idea is to store some of the adjacency relations (possibly in a partial form) explicitly and use them to derive the ones which are not explicitly stored. For example, in the winged-edge representation, each face and vertex point to one of the adjacent edges and each edge – to its endpoints, the two adjacent faces and the four edges adjacent to that edge and the neighboring two faces. Using the above relations it is possible to compute all others (for example, all edges adjacent to a face or all edges adjacent to a vertex) in time proportional to the *local* complexity of the mesh (for our two examples, the number of edges of the face and the number of edges out of a vertex, respectively). One of the concerns of the boundary data structures is to reduce storage space needed to keep adjacency information. However, they take up a lot of space since they also attempt to minimize the time needed to access adjacency information. Therefore, it is not fair to treat boundary data structures as compression schemes. In fact, as shown in [30], such a data structure requires at least $4E$ pointers, where E is the number of edges of the mesh – more than a tetrahedron table.

3.2 Compression Schemes for Tetrahedral and Triangle Meshes

Staadt and Gross [24] and Trotts et al. [28] independently propose a tetrahedral mesh simplification process, which removes tetrahedra by collapsing their edges in a sequence that attempts to minimize, at each stage, the error computed using different cost functions. Such a simplification may be viewed as a lossy compression technique and complements our loss-less compression, which may be used to compactly encode the simplified meshes.

We are not aware of any other work in compressing tetrahedral meshes. However, several approaches that have been proposed for compressing triangle meshes in 2D or 3D could inspire new approaches for compressing tetrahedra.

Deering's approach [4] is a compromise between a standard triangle strip and a general scheme for referencing any previously decoded vertex. Deering uses a 16 register cache to store temporarily 16 of the previously decoded vertices for subsequent uses. He suggests to use one bit per vertex to indicate whether a newly decoded vertex should be saved in the cache. Two bits per vertex are used to indicate how to form a triangle. One bit per triangle indicates whether the next vertex should be read from the input stream or retrieved from the cache. 4 bits of address allow random access of a vertex in the stack-buffer every time an old vertex is reused. One could envision extending the notion of a triangle strip to tetrahedra. Keeping 3 registers for the last 3 vertices used, each new tetrahedron will be defined by these 3 vertices and a fourth vertex either new (the next vertex received in the compressed input stream) or previously received (and identified by its location or id in main memory or cache). One of the vertices in the registers will be replaced by the forth one and the operation repeated. Unfortunately, we do not know of simple and efficient algorithms for identifying the suitable sequence of tetrahedra.

Hoppe's Progressive Meshes [11] permit to transfer a 3D mesh progressively, starting from a coarse mesh and then inserting new vertices one by one. Instead of a vertex insertion to split a single triangle, as suggested in [6] for convex polyhedra, Hoppe applies a vertex insertion that is the inverse of the edge collapse operation popular in mesh simplification techniques [12], [19], [10]. A vertex insertion identifies a vertex v and two of its incident edges. It cuts the mesh open at these edges and fills the hole with two triangles. The vertex v is thus split into two vertices. Each vertex is transferred only once in the Hoppe's scheme.

Hoppe suggests that it may be possible to extend this scheme to tetrahedra. Each vertex split would require identifying one vertex of the current (simplified) version of the mesh and a cone of incident triangles. As the vertex is extruded into an edge, these triangles would be extruded into new tetrahedra. The cost of this approach is the identification of each vertex ($\log_2 v$ per vertex) and the identification of the cone of incident edges, which on average would require 15 bits per vertex. Although simple, this approach would require roughly $(3\log_2|V| + 45)/20$ bits per tetrahedron.

The Topological Surgery method recently developed by Taubin and Rossignac [26] also builds a vertex spanning tree of T that splits the surface of the mesh into a binary tree of corridors (generalized triangle strips). The two trees are encoded using a run length code, which for highly complex meshes yields an average of less than two bits per triangle. In addition, one bit per triangle is used to indicate whether the next triangle in the corridor is attached to the left or to the right edge of the previous one. The compactness of the encoding of both trees comes from the fact that, by construction, both trees tend to have very few nodes with more than one child. Sequences of consecutive nodes with a single child are grouped into runs and encoded by simply storing their length. For pathological cases, with a non-negligible proportion of multi-child nodes, the above approach no longer guarantees linear storage cost. The vertices are stored in depth-first traversal order of the vertex spanning tree. The entire mesh is represented by the list of vertex coordinates, an encoding of the sparse vertex and corridor trees and the string of left/right bits. The application of this technique for VRML files is discussed in [27]. Taubin and Rossignac's technique could be extended to tetrahedral meshes by encoding the tetrahedron spanning tree (as we do) and then by encoding the boundary and the 'cut' which is a two dimensional non-manifold triangulated surface. In some sense, our folding scheme offers a compact encoding of this surface.

Inspired by [15] and improving on [18],[23], Denny and Sohler proposed a technique for compressing *planar* triangulations of sufficiently large size as a permutation of its vertices [5]. They show that there are less than $2^{8.2|V|+O(\log_2|V|)}$ valid triangulations with $|V|$ vertices, and that for sufficiently large $|V|$, each triangulation may be associated with a different permutation of these vertices (there are certainly more than $2^{|V|\log_2(|V|/2)/2}$ of such permutations). Their approach requires transmitting an auxiliary triangle that contains all the vertices and the vertices themselves in a suitable order computed by the compression algorithm. The decoding process sorts V lexicographically and then sweeps over the progressively refined triangulation, from left to right. At each vertex, the enclosing triangle is identified [16] and the vertex is inserted according to the incidence information derived from the permutation. The vertices of V are transmitted progressively in batches. The successive batches are constructed through repetitive plane sweeps, during which all vertices with degree at most six are removed incrementally and the resulting holes retriangulated. For each point, the information needed to reconstruct the triangulation is encoded in the permutation of the vertices of the batch. The batches are compressed in inverse order. Although for sufficiently complex models the cost of storing the mesh incidence is null, the unstructured order in which the vertices are received and the absence of the incidence graph during their decompression makes it difficult to use predictive techniques for vertex data encoding. We believe that this approach may be directly adapted to tetrahedral meshes. However, as in the 2D case, it will make it difficult to compress the vertex data, because the connectivity of each new vertex is derived from its position and hence cannot be used to estimate the position.

Edgebreaker, introduced by Rossignac [22], allows to compress the connectivity of a triangular mesh using only about 2 bits per triangle. Similarly to Grow&Fold, the compression starts with a depth-first search traversal of the dual graph of the mesh. The traversal is topological, i.e. after a triangle is discovered we first visit the triangle adjacent along its right edge whenever it is possible (in the Grow&Fold scheme, the traversal order is arbitrary). Whenever a new triangle is discovered, it is classified as one of five possible types according to which of its edges are shared with triangles which remain undiscovered. A variable length encoding technique is then applied to encode the sequence of types of triangles encountered during the traversal using about 2 bits per triangle. That sequence of triangle types turns out to be sufficient to reconstruct the original mesh. In principle, the Edgebreaker could be extended to 3D case. However, it will no longer be that simple. For example, sometimes extra information would be needed to encode the offset of the fourth vertex when a new tetrahedron without any new vertices is added to the mesh during decompression. Also, the number of ways in which a removal of a tetrahedron can split the mesh into connected components is considerably larger than in the 2D case. Grow&Fold seems to be a simpler and cleaner alternative.

Other compression schemes for planar graphs and triangulations are discussed in [29] and [14].

4 Overview of the Compressed Format

Our encoding of a tetrahedral mesh consists of two parts:

- The *tetrahedron spanning tree string*, defining a tetrahedron tree – a complex containing all tetrahedra appearing in the encoded mesh and some of the incidence relations.

- The *folding string*, defining how to uncover incidence relations absent from the tetrahedron tree by means of folding and gluing operations.

4.1 The Tetrahedron Spanning Tree String

A tetrahedron tree is a three-dimensional simplicial complex which can be obtained from a single tetrahedron as a result of *growing*, i.e. incrementally applying the operation of attaching a tetrahedron to an external face. As shown in Figure 2, attaching a tetrahedron to an external face with vertices v_0, v_1 and v_2 is equivalent to creating a new vertex w and adding a tetrahedron with vertices v_0, v_1, v_2 and w to the mesh. The tetrahedron tree string stores information about which external faces are *attachable*, i.e. to which of

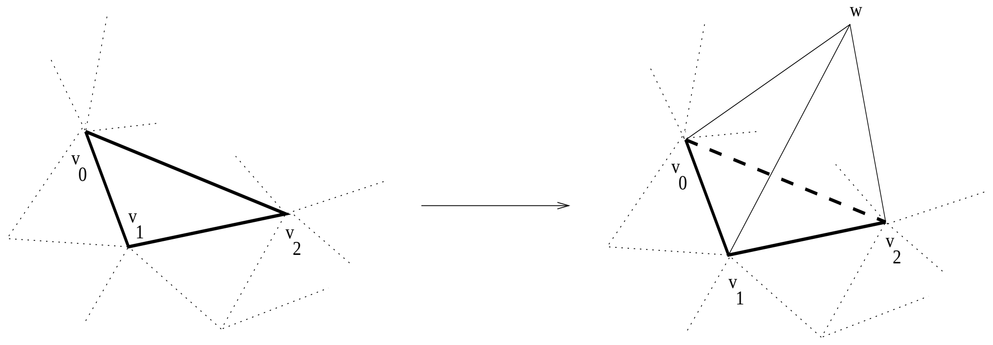

Figure 2: The attaching operation. $v_0v_1v_2$ is an external triangle of the starting mesh and w is a new vertex.

them tetrahedra are attached later in the growing process. Right after each attaching operation, the decompression procedure reads a triple of bits of the encoding string and marks each of the three new external faces (v_0v_1w, v_1v_2w and v_0v_2w on Figure 2) as either attachable or not, according to the value of the corresponding bit of the triple. The tetrahedron tree string consists of one triple of bits per tetrahedron, which yields a total of $3m$ bits.

4.2 Folding String

Using the tetrahedron spanning tree string, the decoding algorithm is able to grow a tetrahedron tree with a tetrahedron table $\mathcal{T}'$, having m rows and referencing $m+3$ vertices where m is the number of tetrahedra in the original mesh $\mathcal{M}$. The tetrahedron tree can be thought of as the result of cutting $\mathcal{M}$ along the surface formed by *cut triangles* (defined in the next section).

Therefore, tetrahedra of the tree correspond to tetrahedra of $\mathcal{M}$ in a one-to-one fashion and each external triangle of the tree either corresponds to an external triangle of $\mathcal{M}$ or belongs to a pair of triangles corresponding to a single cut triangle of $\mathcal{M}$. In order to reconstruct the mesh $\mathcal{M}$ from the tree we must identify the triangles belonging to the same pair. We do it by incrementally applying *gluing* and *folding* operations. A folding operation (Figure 3) 'folds' the boundary of a mesh at an edge. It can be executed only if that edge is the *fold edge* in both external triangles adjacent upon it. As a result, the two incident triangles are identified and become an internal face of the mesh and their two vertices (the ones that bound the two incident triangles but not their common edge) are equated. The folding operation changes the adjacency of nearby faces, so it may make two triangles of the starting mesh which do not share an edge be adjacent along a fold edge. Such triangles are identified by a fold operation later during decompression. Thus, the way in which fold edges are assigned to external triangles (later referred to as the folding scheme) imposes restrictions on the order of execution of folding operations.

The need for the gluing operation, which identifies two arbitrary external triangles, arises when two external triangles do correspond to the same triangle of the mesh $\mathcal{M}$ but never become adjacent along the fold edge in both. Being more general than the folding operation, gluing operation alone suffices to construct $\mathcal{M}$ from the tree. However, the advantage of the folding operation is that it is cheaper to encode.

The folding string associates two bits of information with each external triangle of the tetrahedron tree given by the tetrahedron table $\mathcal{T}'$ except for the one corresponding to the entry face (which is never identified with any other face during decompression). This 2-bit *fold code* distinguishes faces on which the folding operation is to be executed (fold faces) from other faces and, for each fold face, identifies one of its edges as the fold edge. Gluing operations are encoded as two integers identifying the two external triangles to be glued and a two bit code which specifies the 'twist' which has to be applied to one of them before the identification. Thus, gluing operations are considerably more expensive to encode. Fortunately, their number in a typical mesh is relatively small compared to the number of folding operations (for our test cases it was 200-700 times smaller), so that they usually do not contribute to more than 1-2% of the encoding size.

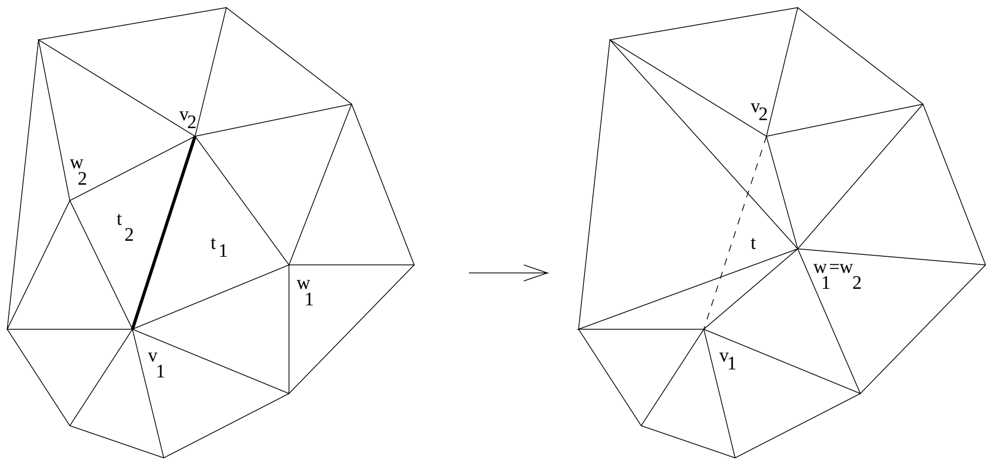

Figure 3: The folding operation as seen from the outside of the mesh. v_1v_2 is the fold edge of both t_1 and t_2 and w_1 and w_2 are the identified vertices. After the identification is done, t_1 and t_2 have the same vertices and therefore become one internal triangle t.

4.3 Compression Results

The total size of our encoding is $7m + 2 + (\lceil \log_2(g + e - 1) \rceil + 1)g$, where m, g and e are the numbers of tetrahedra, glue faces of the tetrahedron tree and external faces of the original mesh. This cost can be broken as follows. The encoding of the tetrahedron spanning tree takes $3m$ bits. Storing the fold codes requires two bits per external face of the tetrahedron tree except for the one corresponding to the entry face. Since a tetrahedron has 4 external faces and attaching a tetrahedron to an external face increases the number of external faces by 2, the total number of external triangles in any tetrahedron tree with m tetrahedra is $2m+2$. It follows that we need $2(2m+1) = 4m+2$ bits to store the fold codes. Fold triangles get nonzero fold codes, while all others (either glue, i.e. identified by means of gluing or corresponding to external faces of the original mesh) get the code of 00. Thus, we can specify a glue face using $\lceil \log_2(g+e-1) \rceil$ bits, where by e and g we denote the number of external faces of the original mesh and glue faces of the tetrahedron tree (respectively). Including the two bit code specifying the twist, each glue triangle pair requires $2\lceil \log_2(g + e - 1) \rceil + 2$ bits to encode. The total size of the encoding of all glue triangle pairs is therefore $(\lceil \log_2(g + e - 1) \rceil + 1)g$

bits.

5 Details of our Approach

5.1 Compression

The compression procedure breaks into three major steps:

1. Building and encoding a tetrahedron spanning tree

2. Creating a folding scheme

3. Building the folding string

The details of each of the above three steps are given below.

5.1.1 Building and Encoding a Tetrahedron Spanning Tree

In order to build a tetrahedron spanning tree one chooses an external triangle of the mesh to be the *entry face* and uses the incident tetrahedron as the root. Starting from the root, we traverse each tetrahedron once using a recursive procedure which systematically selects the next candidate from the undiscovered neighbors of the current tetrahedron. For a tetrahedron which is not the root, by its *door* we mean the triangle which separates it from its parent.

This recursive procedure corresponds to a depth-first search traversal of the dual graph of the mesh, in which nodes correspond to tetrahedra and links to triangles that separate two tetrahedra. Given a tetrahedron spanning tree, there are three types of triangle faces in the mesh.

- external faces (those on the boundary of the mesh),
- doors (triangles corresponding to tree edges of the dual graph of the mesh; equivalently those which are door faces to some tetrahedron),
- cut faces (all others, i.e. internal faces which are not doors).

The encoding of the tetrahedron tree is a sequence of triples of bits, one per tetrahedron, arranged in the traversal order. The i-th bit in a triple encodes whether the i-th face of the corresponding tetrahedron is a door. To make this precise, we need an enumeration order of faces of tetrahedra. Below we discuss how to obtain such an order.

During the traversal, we order vertices of each tetrahedron in the mesh. By convention, the vertices of a tetrahedron are listed in an order v_0, v_1, v_2, v_3 such that v_0, v_1 and v_2 bound its door face (entry face for the root) and define clockwise rotation around the outward pointing normal vector to the tetrahedron at that face. For the root, we choose any ordering such that the first three vertices are vertices of the entry face and orient it clockwise. The order of vertices of a child is determined by that of its parent as follows. Assume that the parent's vertices (in order) are v_0, v_1, v_2, v_3 and let v_4 be the vertex of the child which is not a vertex of the parent. Depending on which three vertices the two tetrahedra share, we assign to the child one of the following orders:

- v_0, v_1, v_3, v_4 if they share v_0, v_1 and v_3,
- v_1, v_2, v_3, v_4 if they share v_1, v_2 and v_3,
- v_2, v_0, v_3, v_4 if they share v_2, v_0 and v_3.

The above rule allows to assign an order of vertices to a tetrahedron right after it is discovered. Note that the two tetrahedra cannot share v_0, v_1 and v_2 because these three vertices bound the door to the parent.

With all this information in hand, we can assign a triple of bits $b_0b_1b_2$ to each tetrahedron as follows. Let v_0, v_1, v_2, v_3 be its vertices (in order). Set b_i to 1 if and only if the face with vertices $v_i, v_{(i+1) \bmod 3}, v_3$ is a door face to some other tetrahedron. In order to obtain the tetrahedron spanning tree string, one needs to concatenate these triples of bits in the traversal order.

We also use the traversal order of tetrahedra to obtain the order in which the vertices have to be rearranged before being made a part of the encoding string. Consider the sequence s of vertices obtained by concatenating sequences of vertices of all tetrahedra in the traversal order (for each single tetrahedron, we always list its vertices in the order assigned during the traversal). Clearly, the length of s is $4m$ and each vertex of the mesh appears as its entry. However, most vertices appear in it more than once (except for those which are vertices of precisely one tetrahedron). To get rid of the repeating vertices, we scan the sequence s, leaving only the entries encountered for the first time and removing all others. The resulting permutation of vertices defines the order in which the vertex data has to be transmitted to ensure correct reconstruction of the mesh geometry by the decompression algorithm.

5.1.2 Creating a Folding Scheme

Recall that the folding scheme imposes restrictions on the order of execution of folding operations so that the decompression procedure restores the structure of the original mesh from the tetrahedron tree. The process of building a folding scheme is essentially an inversion of the gluing and folding operations performed during decompression. This can be seen very clearly when one thinks of it in terms of the complex $\mathcal{C}$ resulting from the original mesh by cutting it along the surface formed by the cut triangles. To construct a folding scheme we delete the cut triangles one at a time. Such a removal of a cut triangle is equivalent to identifying the two external triangles of $\mathcal{C}$ which correspond to that triangle. If the identified triangles share an edge, the identification is a folding operation. This happens if and only if the removed triangle has a *free edge*, i.e. an edge which is an internal edge of the original mesh and, at the same time, is not shared with any other cut triangle. If this is the case, we mark t as an f-triangle (f for fold) and one of its free edges as its f-edge. Otherwise, t is classified as a g-triangle (g for glue). Clearly, the numbers of the f- and g-triangles depend on the order in which they are removed (Figure 4). Since glue triangles cost more and, as we shall see later,

Figure 4: The numbers of g-triangles for the two removal orders are (a) 2, (b) 0. The thick edges are external edges of the mesh. Our greedy strategy leads to the removal order shown in (b), possibly with the third and fourth triangle switched.

their number is twice the number of g-triangles, we would like to make the number of g-triangles as small as we can. In order to do that, we use a greedy strategy: we do not remove cut triangles with no free edges unless there is no other choice.

5.1.3 Building the Folding String

First, we assign two-bit *fold codes* to all pairs (T, t) with T a tetrahedron and t any cut or external triangle (excluding the entry face) adjacent to it. In what follows, we shall call such pairs *cut pairs*. If t is either an external face or a g-triangle, the code is 00. If t is an f-triangle, the code depends on which of its edges is the f-edge. Recall that during the traversal which we did when constructing the tetrahedron spanning tree, we ordered the vertices of each tetrahedron. In particular, we have an ordering v_0, v_1, v_2, v_3 of vertices of T, which permits to introduce an ordering of faces of T. In our implementation, the faces come in order $v_0v_1v_2$, $v_1v_0v_3$, $v_2v_1v_3$, $v_0v_2v_3$. The vertices of these faces are enumerated (relative to T) in the order of being listed above. The ordering of vertices of triangles can be used to define an ordering of their edges (again, relative to T): for a triangle with vertices (in order) w_0,w_1,w_2 its edges come in order w_0w_1,w_1w_2,w_2w_0. The ordering conventions described above are, to some extent, arbitrary, and can be changed without affecting the correctness of our algorithms. Their only special property which will be important later on is that the orders of vertices of faces of a tetrahedron T induce clockwise rotation around its outward pointing normal vectors. Using the orderings, we assign the code of 01, 10 or 11 to the cut pair (T, t) according to whether the f-edge of t is its first, second or third edge.

Apart from the fold code, we need to associate 2 extra bits of information with a g-triangle t. Since it is an internal triangle, there are exactly two tetrahedra T_1 and T_2 adjacent to it. Assuming that T_1 comes before T_2 in the traversal order, the two bit *glue code* simply encodes whether the first vertex of T_1 matches the first, second or third vertex of T_2.

The traversal order together with the orderings of vertices of tetrahedra induce an ordering of cut pairs in the following way. If a tetrahedron T is traversed before T', then any cut pair whose tetrahedron is T precedes any cut pair whose tetrahedron is T'. For cut pairs with the same tetrahedra, i.e. of the form (T, t) and (T, t') we use the ordering of faces of T to break the tie: (T, t) comes before (T, t') if and only if t precedes t' in the ordering of faces of T.

To obtain the folding string, we concatenate:

1. the fold codes of all cut pairs in the above order, obtaining a string of $4m + 2$ bits

2. the encodings of all g-triangles.

The encoding of a g-triangle t consists of:

- The encoding of the two cut pairs having t as their triangle. This requires $2\lceil \log_2(g + e - 1)\rceil$ bits, where e is the number of external faces of the original mesh and g is twice the number of g-triangles (equivalently, the number of glue triangles of the tetrahedron spanning tree reconstructed during decompression). This is because we encode each cut pair as an integer, being the number of cut pairs with the fold code of 00 preceding it in our order and there are $e + g - 1$ such cut pairs.

- the two-bit glue code.

The resulting folding string takes $2(2m + 1) + (\lceil \log_2(g + e - 1)\rceil + 1)g$ bits.

5.2 Decompression

In order to restore the original mesh from its encoding we need to do the following:

1. Grow a tetrahedron tree based on the tetrahedron tree encoding

2. Read and interpret the folding string: classify the external triangles as glue, fold or boundary, assign fold edges to fold triangles, pair up glue triangles and assign a glue code to each pair

3. Initialize datas tructures representing the boundary of the mesh and keeping track of vertex identifications

4. Glue, applying the correct twist determined by the glue code

5. Fold

6. Map the $m + 3$ vertex labels in the tetrahedron tree table into n vertex labels corresponding to vertices of the decoded mesh

5.2.1 Growing a Tetrahedron Tree

```
procedure grow_tree ( s:  bit sequence )
                        :  tetrahedron_table;
var
     t :  tetrahedron_table;
     next_unused_reference,current_bit, i :  integer;
     v0,v1,v2,v3 :  integer; # vertex references
begin
     empty the stack and the table t;
     push(0,1,2);
     next_unused_reference := 3;
     current_bit := 0;
     while current_bit < length(s) do :
          (v0,v1,v2) := pop();
          v3 := next_unused_reference++;
          put (v0,v1,v2,v3) at the end of the table t;
          if s[current_bit+2]=1 then
               push(v2,v0,v3);
          if s[current_bit+1]=1 then
               push(v1,v2,v3);
          if s[current_bit]=1 then
               push(v0,v1,v3);
          current_bit += 3;
end;
```

Figure 5: Growing a tetrahedron tree

The purpose of this part of the decompression algorithm is to build a tetrahedron tree based on the information provided by the tetrahedron tree string and to define the orderings of vertices and tetrahedra consistent with the orderings introduced during compression. The tree growing procedure (whose pseudocode is given in Figure 5) starts with a single tetrahedron and builds a tetrahedron tree by incrementally applying the attaching operation to it. In our implementation, the vertices are represented by integers which can be thought of as vertex labels, which are increasing integers for consecutive vertices as they are first encountered in their construction. A stack is used to keep triples of vertices defining attachable external triangles of the mesh. Initially, the mesh is empty and the stack contains a triple of vertices (represented by the integers 0,1,2) bounding the triangle corresponding to the entry face of the original mesh. The growing procedure pops a list of vertices from the stack and attaches a tetrahedron to the face bounded by those vertices. This is done by creating a new vertex (represented by the least nonnegative integer which has not been used to reference a vertex) and inserting it, together with the three popped vertices, into the tetrahedron table. Then, a triple of bits is read from the encoding string and, for each nonzero bit of that triple, a face of the newly added tetrahedron is pushed on the stack. The way in which the bits of the triple are associated with faces of tetrahedra as well as the order in which the vertices of the pushed triple are listed are the same as used during compression. The output of the growing procedure is a tetrahedron table describing a tetrahedron tree together with an ordering of tetrahedra (given by the order in which they appear in the tetrahedron table) and ordering of vertices for each tetrahedron (the order in which they appear in a corresponding row of the table). Note that the decoding procedure is able to detect where the tetrahedron tree string ends without the need of any separator between it and the folding string.

5.2.2 Reading and Interpreting the Folding String

Although the tetrahedron tree grown based on the tetrahedron tree string contains all tetrahedra and some of the adjacency relations of the original mesh, they do not have the same structure unless the mesh is a tetrahedron tree itself. Geometrically speaking, the tetrahedron tree can be thought of as a result of cutting the original mesh along the surface formed by the cut triangles. A two dimensional example of cutting is shown in Figure 6. Cutting may replicate vertices (in the Figure, the three vertices of the cut marked with '*' are replica of the same vertex of the original mesh). The purpose of folding and gluing is to identify these replicated vertices to a single vertex.

The folding string is used to categorize the external triangles of the tetrahedron tree as fold, glue and boundary, corresponding to f-triangles, g-triangles and external triangles of the original mesh. To each fold triangle, one of its edges is assigned as the fold edge. The glue triangles are paired up and aligned using the two-bit glue code.

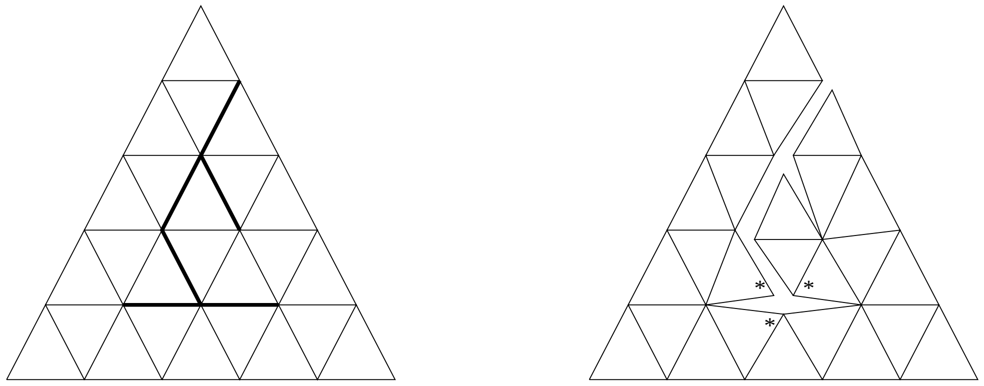

Figure 6: A cut of a two-dimensional mesh along the bold edges; the three vertices of the complex on the right correspond to the same vertex of the original mesh.

This is done by visiting the external faces of the tetrahedron tree in the same order as during compression and using the fold codes in the folding string to identify the face type and the fold edge for fold faces. Faces whose fold code is 01, 10 or 11 become fold faces and have the first second and third edge assigned as the fold edge. Faces with the fold code of 00 become either boundary or glue. Let l be the number of such faces. In order to distinguish boundary triangles from glue triangles we read the g-triangle encodings, which start with the $(4m + 3)$-th bit of the folding string and occupy $2\lceil \log_2 l\rceil + 2$ bits each. Their interpretation is as follows. The first and second $\lceil \log_2 l\rceil$ bits encode two triangles with the fold code of 00, each one of them as an integer being the number of triangles preceding it with that fold code. These two triangles become a pair of glue triangles and they obtain the last two bits of the g-triangle encoding as their associated glue code.

5.2.3 Initialization of the Dastructure Representing the Boundary of the Mesh; Mapping Vertices

The basic building blocks of the representation of the boundary of the mesh are:

- The triangle record, keeping three vertex references (integers, the same as those in the tetrahedron table), three pointers to adjacent edges and a

```
procedure glue;
begin
     for all glue pairs do :
          Let t and u be pointers to the two triangles in the
             pair, ^t precedes ^u in the tetrahedron table
             and v0,v1,v2 and w0,w1,w2 the vertex
             references of these triangles (in order);
          # twist according to the glue code
          if the glue code is 00, (ww0,ww1,ww2):=(w0,w2,w1);
          if the glue code is 10, (ww0,ww1,ww2):=(w2,w1,w0);
          if the glue code is 01, (ww0,ww1,ww2):=(w1,w0,w2);
          update the boundary of the mesh;
          identify pairs of references (v0,ww0), (v1,ww1)
             and (v2,ww2);
end;
```

Figure 7: Gluing procedure

set of flags allowing to determine if the triangle is a fold triangle and, if so, which of its edges is the fold edge.

- The edge record, keeping two pointers to adjacent triangles.

The construction of the above data structure can be implemented as a part of the tree growing procedure: it is initialized it so that it describes the boundary of a single tetrahedron on startup and then updated right after each attaching operation.

Both gluing and folding identify two external triangles of the mesh and therefore change the structure of its boundary. Thus, the data structure storing the boundary of the mesh has to be updated after each glue or fold operation. For a fold operation, an update may be produced by an edge swap followed by an edge collapse (cf [12], [11]). While updating that data structure, we identify the corresponding vertices of the identified triangles. For vertices with labels $i < j$, their identification is equivalent to replacing each occurrence of j in the tetrahedron table by i and subtracting 1 from all labels greater than j. Our actual implementation performs the label changes as a postprocessing step. More precisely, when we glue and fold, we maintain a graph whose vertices are labels $0, 1, \ldots, m+2$ and edges join the identified pairs of labels. After all gluing and folding operations are performed, we compute the mapping of the original labels into target ones by computing and ordering the connected components of the outcoming graph.

5.2.4 Gluing and Folding

We start with performing gluing operations. We go over all glue triangle pairs and identify the two triangles in each pair and their corresponding vertices, updating our representation of the boundary of the mesh. The glue code of each pair provides information about what twist to apply before identifying the two triangles. The pseudocode which performs all the necessary gluing is given on Figure 7.

After all the gluing operations are done, we start folding. Recall that folding along an edge is allowed if and only if that edge is the fold edge of both adjacent external triangles. To avoid scanning edges in search of admissible fold ones, we adopt the following strategy. After any folding operation we recursively attempt to fold along the two external edges of the internal triangle resulting from the folding. In order to do all the folding operations, we call this recursive procedure for all external edges of the mesh.

It can be shown (see [25]) that, at this point, all glue and fold boundary faces have disappeared from the boundary of the mesh (i.e. have been identified with other faces becoming internal triangles). In other words, all exterior triangles of the current mesh are in fact boundary.

6 Complexity

In this section we argue that the compression and decompression algorithms can be implemented so that they run in $O(m\log m)$ and $O(s)$ time (respectively), where m is the number of tetrahedra in the mesh and s is the encoding size.

6.1 Compression

Building a tetrahedron spanning tree requires linear time in the number of tetrahedra, since the dual graph of the mesh has m vertices and $O(m)$ edges.

In order to create a folding scheme, we remove cut triangles one at a time, always removing one with a free edge whenever possible. To implement this process so that it runs in $O(m)$ time one can use a procedure which removes a specified triangle and calls itself recursively for triangles adjacent to those of its edges which become free as a result of that removal. This recursive procedure is first called for all cut triangles with a free edge. After this is done, there are no cut triangles with a free edge left. To get rid of all cut triangles, we simply keep calling the above procedure for an arbitrarily chosen remaining cut triangle (which is then removed and tagged as a g-triangle). Equipping each triangle with an active flag which is reset when the triangle is deleted and storing a count of adjacent active cut triangles for each edge enables to delete cut triangles and test whether an edge is free or not in constant time. Since there are $O(m)$ triangles, the folding scheme can be constructed in $O(m)$ time. The length of the folding string is clearly $O(m\log m)$. Assuming that it takes unit time to write a bit into a string, the process of creating the folding string takes $O(m\log m)$ time. Since the input to the compression procedure is a raw tetrahedron table, it has first to be converted into a dual graph. To do that, one can use a table with $4m$ rows, each of them containing three vertex labels sorted in the increasing order and a pointer to a tetrahedron adjacent to the face bound by these vertices. We build this table by simply scanning the tetrahedra of the mesh and, for each face of the current tetrahedron, putting the labels of its vertices and the pointer to the tetrahedron at the end of the table. This table can then be sorted with respect to lexicographical (or any other) order on triples of vertex labels. By doing that, we put the pointers to adjacent tetrahedra into neighboring rows of the table, therefore making it possible to build the dual graph in linear time (excluding $O(m\log m)$ time spent on sorting). A similar procedure can be used to compute adjacency relations between cut triangles, although this can also be done more efficiently during the mesh traversal.

6.2 Decompression

It takes linear time to build a tetrahedron tree and the data structure representing the mesh boundary (constant time update is necessary for each attaching operation). Reading and interpretation of the folding string takes $O(s)$ time. Each gluing and folding operation can be done in constant time and there are $O(m)$ of them. Thus, folding and gluing takes linear time in m.

Similarly, vertex mapping takes $O(m)$ time since it boils down to computing connected components of a graph with $O(m)$ vertices and edges.

7 Discussion and Open Questions

In this section we discuss future work which may lead to improving the compression ratio achieved by the Grow&Fold algorithm.

First of all, one could encode the glue codes in a more compact way. Instead of using two bits to represent one of the three possible glue codes, it is possible to use only $\lceil \log_2 3^{(g/2)} \rceil = \lceil \frac{g \log_2 3}{2} \rceil$ bits to encode all of them. If the number of glue triangles is large, this leads to savings of over 0.4 bits per glue triangle pair.

Perhaps a more promising (and challenging, too) idea is to look for improvements of the coding scheme for tetrahedron trees. A simple observation that there are exactly $m-1$ bits set to one in our encoding of a tetrahedron tree leads immediately to the conclusion that our coding algorithm for tetrahedron trees is not optimal: for m large, about

$$\lim_{m\to\infty} \frac{\log_2 \binom{3m}{m-1}}{m} = 3\log_2 3 - 2 \approx 2.75$$

bits per tetrahedron are enough to encode such a tree. One may hope for even better results here, since not all sequences with exactly $m-1$ nonzero entries are a valid encodings of tetrahedron trees (any encoding string must have the property that there are at least k entries equal to one among the initial $3k$ symbols for any $k < m$).

Another interesting question concerns the number of glue triangles which are the reason for the nonlinear term in our estimate of the encoding length. Glue triangles are certainly needed for meshes with handles. Moreover, the number of glue triangles cannot be smaller than the number of handles. This is because each handle gives rise to a shell of cut triangles with boundary contained in the boundary of the mesh. No matter what the removal order of cut triangles during compression is, each such shell has to be broken at some point by removing a cut triangle with no free edges. However, even for meshes with no holes or handles glue triangles may be necessary. One can imagine a triangulation of the three dimensional ball and its tetrahedron spanning tree for which the cut triangles form a superset of the house with two rooms ([3, section I.2]): a two dimensional simplicial complex which is contractible but whose triangulation has no triangle with a free edge (Figure 8). If this is the

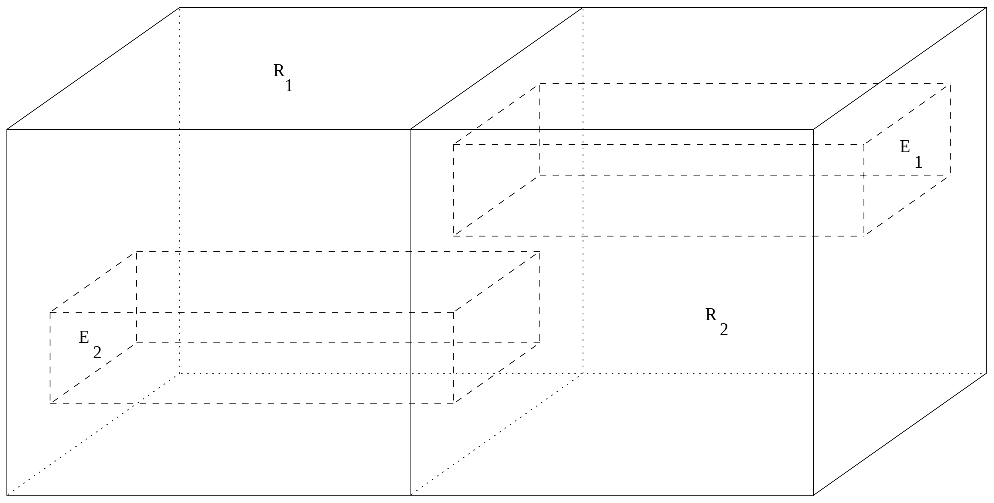

Figure 8: The house with two rooms. It defines two 'rooms' R_1 and R_2. In order to enter R_i one has to walk through the corridor through the other room, starting with the 'door' E_i. The house consists of the walls of the two rooms with the corridors' entrances and exits removed and the corridors' boundaries.

case, at least one glue triangle is needed (because the first triangle removed from the house must not have a free edge). However, it may be possible to change the tetrahedron tree so that no house with two rooms appears in the cut complex. For example, one could 'close' the entrance E_1 in Figure 8 and, simultaneously, remove some triangle from the wall of R_1. The resulting space has the property that triangles can be removed from it one at a time in such a way that each removed triangle has a free edge. We see it as an indication that by changing the cut (or, equivalently, tetrahedron spanning tree) it is possible to decrease the number of glue triangles and therefore improve the performance of our coding algorithm. It would be interesting to develop an efficient algorithm which, by changing the tetrahedron tree, decreases the number of glue triangles (perhaps to the number which is optimal for the input mesh).

n	m	l	l_g	l_g/l	l/m	T_c	T_d
100	514	3628	28	0.008	7.058	0.09	0.04
1000	6298	44430	342	0.008	7.055	1.19	0.52
10000	66487	469657	4246	0.009	7.064	14.99	6.27
50000	335188	2373774	27456	0.012	7.082	80.43	31.93
100000	672212	4767534	62048	0.013	7.092	166.84	70.84

Figure 9: Results of our experiments

8 Experimental results

We tested our algorithm by running its prototype implementation for Delaunay tetrahedralizations of random sets of points in a cube. The tetrahedralizations were generated using the program `qhull` from the Geometry Center of the University of Minnesota. The results are given in Figure 9. In particular, they show that it is quite easy to obtain meshes which require nonzero number of g-triangles. However, the number of such triangles is relatively small, so that the encodings of the glue triangle pairs usually do not contribute to more than 1-2% of the total encoding size. The explanation of symbols used to describe the meaning of the columns of the table in Figure 9 is given below.

l - the total length of the encoding string

l_g - length of the encoding of the glue triangle pairs

n - number of points

m - number of tetrahedra

T_c - running time of the compression algorithm (in seconds)

T_d - running time of the decompression algorithm

The running times are the real time measurements. We ran our implementation on an SGI Power Challenge, with no effort to optimize or parallelize the code. One can notice that the running time growth is close to linear in the number of vertices of the mesh.

9 Conclusion

We discussed a simple topological compression scheme for connectivity of tetrahedral meshes which allows to store it using about 7 bits per tetrahedron. Our scheme can be compared to the standard representation via a tetrahedron-vertex incidence table, which requires $4\lceil \log n \rceil$ bits per tetrahedron, where n is the number of vertices of the mesh. We described efficient compression and decompression algorithms, running in $O(m \log m)$ time (compression) and linear time in the encoding size (decompression), where m is the number of tetrahedra in the mesh. We do not have a linear bound on the encoding size, but the results of experiments with our prototype implementation show that our algorithm produces encodings whose length is nearly linear in m.

References

[1] B.G.Baumgart, Winged Edge Polyhedron Representation, AIM-79, Stanford University, Report STAN-CS-320, 1972.

[2] B.G.Baumgart, A Polyhedron Representation for Computer Vision, *AFIPS Nat. Conf. Proc.*, Vol.44, 589-596, 1975.

[3] M.M.Cohen, *A Course in Simple-Homotopy Theory*, Springer-Verlag 1970.

[4] M.Deering, Geometric Compression, *Computer Graphics (Proc. SIGGRAPH),* p.13-20, August 1995.

[5] M.Denny and C.Sohler, Encoding a triangulation as a permutation of its point set, *Proc.9th Canadian Conference on Computational Geometry*, pp.39-43, Ontario, August 11-14, 1997.

[6] D.Dobkin and D.Kirkpatrick, A linear algorithm for determining the separation of convex polyhedra, Journal of Algorithms, Vol.6, pp.381-392, 1985.

[7] L.Floriani and B.Falcidieno, A Hierarchical Boundary Model for Solid Object Representation, ACM Transactions on Graphics 7(1), pp.42-60, 1988.

[8] E.Gursoz and F.Prinz, Boolean Set Operators on Non-Manifold Boundary Representation Objects, Computer-Aided Design 23(1), pp.33-39, January/February 1991.

[9] E.Gursoz, Y.Choi and F.Prinz, Node-Based Representation of Non-Manifold Surface Boundaries in Geometric Modeling, In: J.Turner, M.Wozny and K.Preiss eds., *Geometric Modeling for Product Engineering*, North-Holland 1989.

[10] P.Heckbert and M.Garland, Survey of Polygonal Surface Simplification Algorithms, in *Multiresolution Surface Modeling Course*, ACM SIGGRAPH Course Notes, 1997.

[11] H.Hoppe, Progressive Meshes, *Computer Graphics (Proc. SIGGRAPH),* p.99-108, August 1996.

[12] H.Hoppe, T.DeRose, T.Duchamp, J.McDonald and W.Stuetzle, Mesh Optimization, *Computer Graphics (Proc. SIGGRAPH),* p.19-26, August 1993.

[13] Y.E.Kalay, The Hybrid Edge: A Topological Data Structure for Vertically Integrated Geometric Modeling, Computer-Aided Design 21(3), pp.130-140, 1989.

[14] K.Keeler and J.Westbrook, Short Encodings of Planar Graphs and Maps, Discrete Applied Mathematics, No. 58, pp.239-252, 1995.

[15] D.Kirkpatrick, Optimal search in planar subdivisions, SIAM Journal of Computing, vol 12, pp 28-35, 1983.

[16] D.T.Lee and F.P.Preparata, Location of a point in a planar subdivision and its applications, SIAM Journal on Computing, Vol.6, pp.594-606, 1977.

[17] M.Mäntylä, *An Introduction to Solid Modeling*, Computer Science Press, Rockville, Maryland 1988.

[18] M.Naor, Succinct representation of general unlabeled graphs, Discrete Applied Mathematics, vol. 29, pp. 303-307, North Holland, 1990.

[19] R.Ronfard and J.Rossignac, Full-range approximation of triangulated polyhedra, *Proc. Eurographics'96, Computer Graphics Forum,* pp. C-67, vol.15, no.3, August 1996.

[20] J. Rossignac and M. O'Connor, SGC: A Dimension-independent Model for Pointsets with Internal Structures and Incomplete Boundaries, in Geometric Modeling for Product Engineering, Eds. M. Wosny, J. Turner, K. Preiss, North-Holland, pp. 145-180, 1989.

[21] J.Rossignac, Through the cracks of the solid modeling milestone, *From Object Modeling to Advanced Visual Communication,* Eds. S.Coquillart, W.Strasser, P.Stucki, Springer-Verlag, pp. 1-75, 1994.

[22] J.Rossignac, Edgebreaker: Compressing the connectivity of triangle meshes, GVU Technical Report GIT-GVU-98-17, Georgia Institute of Technology, `http://www.cc.gatech.edu/gvu/reports/1998`.

[23] J.Snoeyink and M.van Kerveld, Good orders for incremental (re)construction, Proc. ACM Symposium on Computational Geometry, pp.400-402, Nice, France, June 1997.

[24] O. Staadt and M. Gross, Progressive Tetrahedralization, Proc. IEEE Visualization, pp. 397:402, Research Triangle Park, October 18-23, 1998.

[25] A.Szymczak and J.Rossignac, Grow & Fold: Compression of Tetrahedral Meshes, GVU Technical Report GIT-GVU-98.

[26] G.Taubin and J.Rossignac, Geometric Compression Through Topological Surgery, ACM Transactions on Graphics, Vol.17, no.2, pp.84-115, April 1998.

[27] G.Taubin, W.Horn, F.Lazarus and J.Rossignac, Geometry Coding and VRML, Proceedings of the IEEE, pp.1228-1243, vol.96, no.6, June 1998.

[28] I. Trotts, B. Hamann, K. Joy, D. Wiley, Simplification of Tetrahedral Meshes, Proc. IEEE Visualization, pp. 287:295, Research Triangle Park, October 18-23, 1998.

[29] G.Turan, On the Succint Representation of Graphs, Discrete Applied Mathematics 8, pp.289-294, 1984.

[30] T.C.Woo, A Combinatorial Analysis of Boundary Data Structure, IEEE Computer Graphics and Applications, Vol.5, pp.19-27, 1985.