Facebreaker: A Compression for Tetrahedral Meshes without Boundary

John DeIonno[1], Anna Shustrova[2], Ives José de Albuquerque Macêda Júnior[3]

[1]University of California, Los Angeles – USA
[2]University of California, Berkeley – USA
[3]Pernambuco Federal University – Informatics Center – Brazil

Abstract

Tetrahedral meshes used in applications such as volume visualization consume very large amounts of memory. Thus compression is essential for storage and transmission purposes. A lot of different schemes have been developed to address this problem. Our algorithm handles tetrahedral meshes that are connected oriented combinatorial 3-manifolds without boundaries. It extends the Edgebreaker algorithm on a Corner Table for triangular meshes to work with tetrahedral meshes.

Keywords: tetrahedral meshes; compression algorithms; Edgebreaker algorithm; corner table;

1. Introduction

Mesh compression techniques are a part of a new branch of modern data compression techniques, 3D compression. In the case of tetrahedral mesh compression, the main areas of interest include but are not limited to simulations on volumetric domains and volume visualization. In most applications there is some data attached to the elements of the tetrahedral mesh. This data can be attached to the vertices, edges, faces or tetrahedra [1]. Some of the applications are: Digital Entertainment, Computer Aided Design (CAD), Computer Aided Manufacturing (CAM), Computer Aided Engineering (CAE) and Computational Tomography (CT). All these areas use very large meshes and thus have great demands for data storage and transmission. A good compression scheme for tetrahedral meshes is therefore of an utmost importance.

Our approach to mesh compression deals only with tetrahedral meshes that are connected oriented combinatorial 3-manifolds. In simple terms we require the following condition to hold: each face in the mesh is incident to one or two tetrahedra. In particular our algorithm can only work with meshes without boundaries. This changes the condition to: each face is incident to exactly two tetrahedra.

The approach we present in this paper extends the Edgebreaker algorithm for triangular meshes [2] and consists of traversing tetrahedral meshes from a tetrahedron to an adjacent tetrahedron in order to compress and decompress it. As we traverse the mesh, our compression method produces a code that describes the topological relations of the current tetrahedron to the rest of the mesh. Decompression uses these codes and corner table data structure to infer connectivity among tetrahedrons in order to reconstruct the mesh.

2. Orientation, Conventions and Cases

When traversing a tetrahedral mesh a move is made through the face of the current tetrahedron to an adjacent tetrahedron that has not been visited yet, and then the new tetrahedron

and all its vertices are marked as visited. To traverse tetrahedral meshes in an orderly manner it is essential to first create a convention for marking direction. We mark the faces on each tetrahedron as *source*, *top*, *right* and *left*. To do so it is necessary to keep track of the last two moves. In particular, we keep track of the vertex *w* opposite to the face we went through on the previous move. Given this information, we say the face we just went through is the *source* and the face opposite to *w* is the *top*. *Left* and *right* faces are determined as shown in *Figure 1*, where we are marking faces for the tetrahedron in the center and the previous tetrahedron (not shown) is above the paper. As a convention we always go through the *top* if the incident tetrahedron is

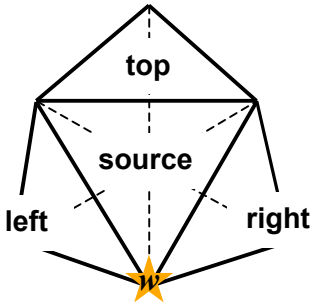


Figure 1: Face Marking Convention

not visited, else we try going *right* and only if both top and right tetrahedrons are visited we go through the *left*.

We next define the cases that can come up when traversing a tetrahedral mesh without boundaries. The nine cases, denoted as C, T, R, ~T, ~R, ~L, E and S, are distinguished according to *Table1*, where *v* is the vertex opposite to the *source* face of the current tetrahedron and Top, Right and Left correspond to tetrahedrons adjacent to the respective faces of the current tetrahedron. *Figure 2* illustrates the nine cases, where blue shading indicates that tetrahedron has already been visited.

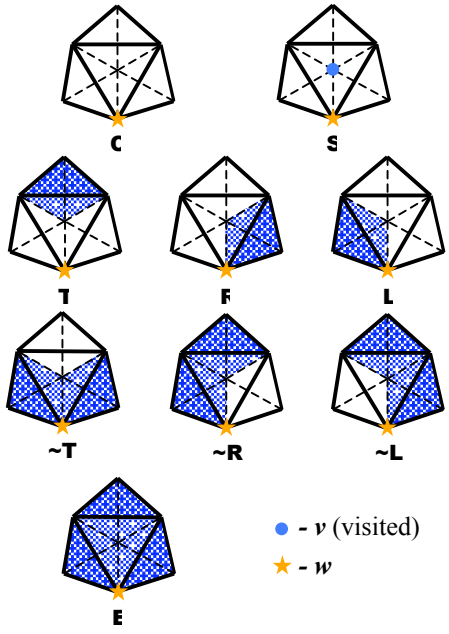


Figure 2: Facebreaker cases illustrated

	v	Top	Right	Left
C	not visited	not visited	not visited	not visited
T	visited	visited	not visited	not visited
R	visited	not visited	visited	not visited
L	visited	not visited	not visited	visited
~T	visited	not visited	visited	visited
~R	visited	visited	not visited	visited
~L	visited	visited	visited	not visited
E	visited	visited	visited	visited
S	visited	not visited	not visited	not visited

Table 1: Facebreaker states

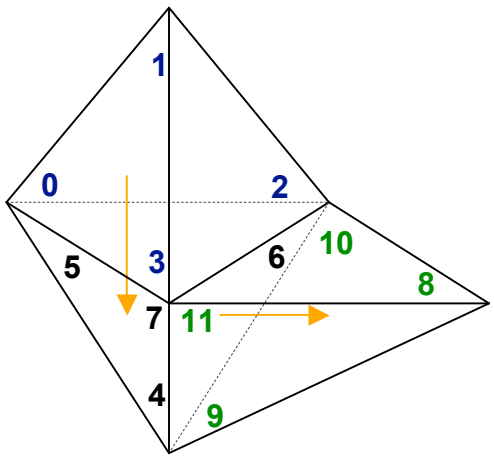
3. Corner Table

Originally the Corner Table was used as a very compact data structure for storing connectivity information for triangular meshes [2]. We extend this structure to work for tetrahedral meshes. A *corner* represents the association of a tetrahedron with one of its incident vertices. The idea is that each corner has a vertex and an opposite corner associated to it. By opposite corners we mean two corners in adjacent tetrahedrons that are not incident to the face between these tetrahedrons. The information for each corner's opposites and their incident vertices can be stored in two integer arrays called O and V, respectively. The dimension of both arrays is equal to number of tetrahedra times four.

Our convention for numbering corners in a tetrahedron is as follows: suppose we just moved to a new tetrahedron and the number for the last corner marked was n–1. The corner incident to v is numbered n, the corner incident to w is numbered n+1, the order for numbering the last two corners is determined by the right hand rule with the thumb pointing from w to v. Note that this way the corner n is opposite to the *source*, n+1 – to the *top*, n+2 and n+3 – to the *right* and *left*, respectively. For convenience we will refer to corner n as the **source** corner, n+1 as the **top**, e*tc*.

4. Initial Conditions

To start the traversal of a tetrahedral mesh we pick an arbitrary tetrahedron and arbitrarily mark corners 0 and 1 and then use the right hand rule to mark 2 and 3 (*Diagram 3*). We also label their incident vertices 0 through 3 accordingly. We then make a move though the face opposite to vertex 1. The corner and its incident vertex opposite to the *source* of the newly discovered tetrahedron are labeled 4. The corner that is incident to vertex 1 is labeled as 5, the corner incident to 2 as 6 and the corner incident to 3 as 7. The last arbitrary move we make is through the face opposite to vertex 0. *Table 2* summarizes corner table initialization procedure. We now have both vertices v and w to start labeling corners its incident vertices and faces of all other tetrahedra using the conventions described in previous sections.



		O	V
source	0		0
top	1	4	1
right	2		2
left	3		3
source	4	1	4
top	5	8	0
right	6		2
left	7		3
source	8	5	
top	9		4
right	10		2
left	11		3

Figure 3: First Two moves Illustrated *Table 2: Corner TableInitialization*

5. Compression

The compression algorithm takes in a list of 4-tuples, where an element of a tuple is a vertex number, a 4-tuple is a tetrahedron represented by its 4 vertices and the whole list represents a tetrahedral mesh. It returns a string of symbols from the set {C, T, R, L, ~T, ~R, ~L, E, S} encoded using binary format {0, 1100, 1010, 1001, 1011, 1101, 1110, 1111, 1000}. This string is a very compact way of storing topological connectivity of a tetrahedral mesh.

We start the traversal (see Section 4) by picking an arbitrary tetrahedron from the mesh and relabling its vertices from 0 to 4 in the same way as we labeled the corners. We then make our two initial moves again relabeling the vertices. We keep track of our relabelings in two arrays A[] and N[] that have demension equal to the number of vertices. A[] contains the old vertices as they correspond to the new and N[] contains the new as they correspond to the old.

After the initial moves the algorithm makes a decision to move from the current tetrahedron to one of the adjacent ones. This decision will depend on the type of the current tetrahedron (defined in Section 2). If the current tetrahedron is of type T for example the Top tetrahedron has already been visited and based on our movement order convention we chose to move to the Right tetrahedron.

When dealing with very large meshes the decompression algorithm can theoretically run into problems in which it would have multiple options for reconnecting tetrahedra. So far we have no way of predicting when those problems can occur and thus we need to simulate decompression while we compress the mesh. Therefore, it would make more sense to talk about the compression algorithm in detail after we discuss the decompression.

Before we go on to the decompression algorithm, it is necessary to mention the putty list. Whenever the decompression simulated during compression runs into a problem we append the missing information (a vertex) to this list. During the real decompression when a problem arises we fall back to the putty list. More details about the uses of the putty and also its construction will be given in the next two sections.

6. The Decompression Algorithm

The decompression algorithm builds the corner table by processing the information stored in the *compressed mesh* file. In the *initDecomp* procedure, the algorithm first creates a corner table's *O*[] and *V*[] arrays based on the number of tetrahedra in the mesh and then initializes it as described in section 4 to account for the first two tetrahedra not stored in the *compressed mesh*. *initDecomp* also initializes next new vertex (*NV*) to 5, *current* tetrahedron to 2 and putty list *index* to 0. The algorithm's *decomp* procedure continues to fill in the corner table by successively attaching a new tetrahedron (*current*) to the *previous* tetrahedron and extracting other connectivity information for the *current* depending on its type. The type of each new tetrahedron is obtained from the codes contained in the *compressed mesh* file. *Decomp*() stops when there are no more codes left. The steps taken to fill in the corner table for each type of tetrahedron are as follows.

1. Tetrahedron C (binary code 0): The SOURCE (current tetrahedron's **source**) corner is incident to a new vertex, which gives us *V*[SOURCE] = *NV*, increment *NV* by 1. Case C

means we can move to Top, Right and Left tetrahedra. Based on our convention we move to the Top tetrahedron – *decomp*() calls the *moveToTop*() procedure.

The procedure *moveToTop*() sets the opposite corner of the TOP corner to (SOURCE + 4) — the next tetrahedron's **source** and *O*[SOURCE + 4] = TOP. It also identifies the vertices incident to the next tetrahedron's **top**, **right** and **left** corners and the SOURCE, RIGHT, LEFT respectively as the same. Thus *V*[TOP + 4] = SOURCE, *et*.

2. Tetrahedron T (binary code 1100): Case T means the Top has already been visited and so the *top* face is connected to a face of some previously visited tetrahedron. *decomp*() calls the *glueTop*() procedure to infer this connectivity. With the Top visited we can still move to Right and Left, by convention we move to Right – *decomp*() the *moveToRight*() procedure.

   The procedure *glueTop*() looks for a face to glue the *top* face of the *current* tetrahedron. This face is defined by the vertices R and L incident to RIGHT and LEFT corners and the unknown vertex S of the SOURCE corner. The procedure uses the known edge LR on the face to be glued and the *gluable*(tetrahedron, LR, output S, output U) function called on tetrahedra starting 2 tetrahedra back from the *current* to find the correct face to glue. Once a face is found, the procedure continues to look for other possible faces. If no other faces are found the procedure glues the face found to the *top* face of the *current* tetrahedron. The connectivity information inferred from this action is as follows. *V*[SOURCE] is set to S, the *O*[TOP] is set to U and *O*[U] to TOP, where S is the third vertex on the glued faces and U is the incident corner of the remaining vertex in the found tetrahedron. If however another *gluable* face is found then the procedure reads in the connectivity information above from the *putty* list.

   The function *gluable*(T, *edge*, output S, output U) takes in the index of a tetrahedron T and an edge and returns true if T has a *gluable* face and false otherwise. For a face to be *gluable* it has to contain the same *edge* as the face to be glued on the *current* tetrahedron, the opposite of the corner incident to the other vertex in the face has yet no value assigned and the face has to be oriented correctly. Here we are orienting the faces about the *edge* all tetrahedrons that have a possible *gluable* face revolve about. We put the south and north poles at the vertices of the *edge* and define the two faces on a tetrahedron that share this *edge* as west and east. So if the face on the *current* is facing west and the face on T is also facing west it is not *gluable*.

   The procedure *moveToRight*() sets the opposite of the RIGHT corner to (SOURCE + 4) and *O*[SOURCE + 4] = RIGHT. It also identifies the vertices incident to the next tetrahedron's **top**, **right** and **left** corners and the SOURCE, LEFT, TOP respectively as the same.

3. Tetrahedron R (binary code 1010): Case R means the Right tetrahedron has already been visited and so the *right* face can be glued to a face of a previously visited tetrahedron using the *glueRight*() procedure. Since the Top has not been visited *moveToTop*() is called.

   Procedure *glueRight*() work just like *glueTop*(), except that it looks for a face to glue the *right* face of the *current* tetrahedron which is defined by the vertices T and L incident to TOP and LEFT corners and the unknown vertex S of the SOURCE corner. If only one *gluable* face is found the connectivity information for the SOURCE vertex is the same as in *glueTop*() but for

the opposites we get *O*[RIGHT] is set to U and *O*[U] to RIGHT. If however another *gluable* face is found we again fall back to the *putty* list.

4. Tetrahedron L (binary code 1001): Case L means the Left has been visited and so the *left* is glued with the help of *glueLeft*() procedure. Again, since the Top has not been visited *moveToTop*() is called.

   The procedure *glueLeft*() is like the other glue procedure discussed above. Here we look at the edge defined by vertices incident to RIGHT and TOP corners. The inferred connectivity is again the same except that we find the opposite for the LEFT corner.

5. Tetrahedron ~T (binary code 1011): In case ~T the LEFT and RIGHT tetrahedra a marked as visited. This gives us two faces to glue — *right* and *left*. *decomp*() first calls *glueRight*() and then the *pasteLeft*() procedure to infer connectivity. And finaly, *moveToTop*() is called.

   The prrocedure *pasteLeft*() capitalizes on the fact that the vertex incident to the SOURCE has been discovered earlier in a *glue__*() procedure called right before and so all three vertices of the face to be glued are known. *pasteLeft*() checks each tetrahedron starting 2 tetrahedra back from *current* if they contain the same vertices and the vertices incident to the SOURCE, TOP, and RIGHT corners. Since two and only two tetrahedra can share the same face the search is stopped once one is found. We now have the opposite for the LEFT equals the corner opposite to the pasted face of the second tetrahedron.

6. Tetrahedron ~R (binary code 1101): In case ~R the *top* and *left* faces can be glued with *glueTop*() and *pasteLeft*, respectively. Finally we move to the Right tetrahedron and so *decomp*() calls *moveToRight*().

7. Tetrahedron ~L (binary code 1110): Since the Top and Right have been visited the *top* and *right* faces can be glued with *glueTop*() and *pasteRight*, respectively. And the only tetrahedron we can move to is the Left – call *moveLeft*().

   The procedure *pasteRight*() is similar to *pasteLeft*(). The only difference is the face we are searching for is defined by the vertices incident to the SOURCE, TOP, and LEFT corners.

8. Tetrahedron E (binary code 1111): If a tetrahedron is of type E all the tetrahedra connected to its faces have been visited before. First, *decomp*() calls the *glueTop*(), *pasteRight*() and *pasteLeft*() procedures. Since there are no possible tetrahedra to move to next *decomp*() calls *popBack*() to find a tetrahedron among previously visited tetrahedra that still has an unvisited tetrahedron connected to it.

   PopBack() looks for the first possible tetrahedron going back consecutively from the current tetrahedron that is adjacent to a tetrahedron that has not yet been visited and pops back there. It does so by checking all corners of each tetrahedron and stops at the first one that has at least one corner that still doesn't have an opposite. The opposite of that corner is then set to (SOURCE +4) and vise versa. If the tetrahedron we popped back to has more then one

corners with unknown opposites we use same convention as when deciding which tetrahedron to move to next.

9. Tetrahedron S (binary code 1000): In the type S tetrahedron vertex *v* is visited, but there are no visited tetrahedra adjacent to it, except the Source. Thus there is no information to obtain *V*[SOURCE]. *Decomp*() gets this vertex from the putty list.

Decomp() completely fills in the corner table. No other manipulations on the table are need.

7. The Compression Algorithm

The compression algorithm starts with calls to *initComp*() that performs the steps discussed in Section 5 and to *initDecopm*() to initialize the corner table for the decompression simulation.

The procedure *compress*() is then called recursively until all the tetrahedra in the mesh are marked as visited. *Compress*() checks each tetrahedron starting with the third one against the nine possible cases and appends the correct one to the *codes* string. Once the correct case has been determined *decompSim*() is called to that filled in the information in the corner table according to the case. The breakdown of information obtainable from each case is discussed in Section 6.

When simulating the decompression of a case the algorithm runs into a problem when there is more than one face to glue to the face of the *current* tetrahedron. This is when the *putty* list comes into play. It is sufficient to append the vertex incident to the SOURCE corner to the *putty* list for use during the real decompression. Another case where the use of the *putty* list is necessary is the S case and again only the vertex incident to the SOURCE is sent.

After compressing and decompressing a tetrahedron *compress*() moves to the next one according to convention. When a tetrahedron of type E is reached, as in decompression, the algorithm looks for the first possible tetrahedron going back consecutively from the current tetrahedron that is adjacent to one or more tetrahedra that have not yet been visited and pops back there. The next move is made from that tetrahedron to one of the unvisited ones again by convention.

The compression is finished when all of the mesh has been successfully compresses into a *codes* string and the *putty* list.

9. Improvements to the Algorithm

Compared to the amount of memory required to store each code the vertices stored in the putty list take much more memory. Thus having a large putty list undermines the efficiency of mesh compression into codes. There are a lot of possibilities for improvement, the goal being elimination of the putty list.

Perhaps the most entries in the putty list are caused by the S case. We've considered two possible ways of dealing with S cases. One way attempted to derive the missing vertex by looking at all vertices discovered before a particular S case comes up and eliminating all that cannot possibly be this vertex. For example, we can automatically eliminate the three known

vertices of that S tetrahedron and the fourth vertex of the tetrahedron right before. Unfortunately, we were only able to narrow the list of possible vertices by about two thirds.

The second approach tried to reduce the number of S cases or in the best case scenario remove them completely. Whenever an S case was reached we tracked back to the previous tetrahedron and moved in a different direction if possible. This approach reduced and in simpler case removed S's altogether.

We also tried to improve on the number of vertices appended to the putty list due to multiple 'to be glued to' faces. The approach we now use creates a new code "." to be appended to the string right after the case in question when the correct *gluable* face is not the first one found. We can then append another "." if the correct face is not the second one either and so on. This approach completely eliminates gluing problem vertices from the putty list.

10. Acknowledgments

This work was funded by NSF-USA (INT-0306998) and CNPq-Brazil as part of REU at PUC-RJ, Brazil.

We would like to thanks the Mathematics Department of PUC for their hospitality, our faculty advisor Hélio Lopes, and REU organizers Maria Helena Noronha and Carlos Tomei.

11. References

[1] Stefan Grumhold, Stefan Guthe, Wolfgang Straßer. Tetrahedral Mesh Compression with the Cut-Border Machine.

[2] J. Rossignac, A. Safanova, A. Szymczak. 3D compression made simple: Edgebreaker on a Corner Table. Shape Modeling International Conference, Genoa, Italy May 2001.