Invited lecture at the Shape Modeling International Conference, Gemoa, Italy. May 7-11, 2001.

3D Compression Made Simple: Edgebreaker on a Corner-Table

Jarek Rossignac, Alla Safonova, Andrzej Szymczak
College of Computing and GVU Center
Georgia Institute of Technology
<jarek@cc.gatech.edu, asafonov@cc.gatech.edu, andrzej@cc.gatech.edu>

Abstract

Edgebreaker is a simple technique for compressing 3D triangle meshes. We introduce here a new formulation, which leads to a simple implementation. We describe it in terms of a simple data structure, the Corner Table, which represents the connectivity of any manifold triangle mesh as two table of integers. For meshes that are homeomorphic to a sphere, Edgebreaker encodes these two tables with less than 2 bits per triangle. It compresses vertex locations using a parallelogram predictor. Entropy encoding reduces this cost in practice to less than a bit per triangle when the mesh is large. The detailed source code for the compression and the decompression algorithms fits on a single page. Through minor modifications, the Edgebreaker algorithm has been adapted to manifold meshes with holes and handles, to non-triangle meshes, and to non-manifold meshes. A Corner-Table implementation of these extensions will be described elsewhere.

1. Introduction

3D graphics plays an increasingly important role in applications where 3D models are accessed through the Internet. Due to improved design and model acquisition tools, to the wider acceptance of this technology, and to the need for higher accuracy, the number and complexity of these models are growing more rapidly than phone and network bandwidth. Consequently, it is imperative to continue increasing the terseness of 3D data transmission formats and the performance and reliability of the associated compression and decompression algorithms.

Although many representations have been proposed for 3D models, polygon and triangle meshes are the de facto standard for exchanging and viewing 3D models. A triangle mesh may be represented by its vertex data and by its connectivity. *Vertex data* comprises coordinates of all the vertices and optionally the vertex colors and the coordinates of the associated normal vectors and textures. In its simplest form, *connectivity* captures the incidence relation between the triangles of the mesh and their bounding vertices. It may be represented by a triangle-vertex incidence table, which associates with each triangle the references to its three bounding vertices.

In practice, the number of triangles is roughly twice the number of vertices. Consequently, when pointers or integer indices are used as vertex-references and when floating point coordinates are used to encode vertex locations, uncompressed connectivity data consumes twice more storage than vertex coordinates.

Vertex coordinates may be compressed through various forms of vector quantization. Most vertex compression approaches exploit the coherence in vertex locations by using local or global predictors to encode corrections instead of absolute vertex data. Both the encoder and the decoder use the same prediction formula. The encoder transmits the difference between the predicted and the correct vertex data. It uses variable length codes for the corrections. The better the prediction, the shorter the codes. The decoder receives the correction, decodes it and adds it to the predicted data to obtain the correct information for the next vertex. Thus the prediction can only exploit data that has been previously received and decoded. Most predictive schemes require only local connectivity between the next vertex and its previously decoded neighbors. Some global predictors require having the connectivity of the entire mesh. Thus it is imperative to optimize connectivity compression techniques that are independent of vertex data.

The Edgebreaker compression scheme discussed here has been extended to manifold meshes with handles and holes [Ross99], to triangulated boundaries of non-manifold solids [RoCa99], and to meshes that contain only quadrilaterals or a combination of simply-connected polygonal faces with an arbitrary number of sides [King99b]. It was also optimized for meshes with nearly regular connectivity [SKR00, SKR00b]. Nevertheless, for sake of simplicity, in this paper, we restrict our focus to meshes that are each homeomorphic to a sphere.

As several other compression schemes [TaRo98, ToGo98, Gust98], Edgebreaker visits the triangles in a spiraling (depth-first) triangle-spanning-tree order and generates a string of descriptors, one per triangle, which indicate how the mesh can be rebuilt by attaching new triangles to previously reconstructed ones. The popularity of Edgebreaker lies in the fact that all descriptors are symbols from the set {C,L,E,R,S}. No other parameter is needed. Because half of the descriptors are Cs, a trivial code (C=0, L=110, E=111, R=101, S=100) guarantees 2 bits per triangle. A slightly more complex code guarantees 1.73 bits per triangle [King99]. This upper-bound on storage does not rely on statistic-based entropy or arithmetic coding schemes, which in general perform poorly on small or irregular meshes. Consequently, Edgebreaker is particularly attractive for compressing large catalogs of small models. For large meshes, entropy codes further reduce the storage to less than a bit per triangle [RoSz99]. The string of descriptors produced by Edgebreaker is called the *clers* string. *(No relation with any of the strings pulled by Claire.)* An efficient decompression algorithm for the *clers* sequence [RoSz99] interprets the symbols to build a simply connected triangulated polygon, which represents the triangle-spanning tree. Then, it zips up the borders of that polygon by matching pairs of its bounding edges in a bottom-up order with respect to the vertex-spanning-tree that is the dual of the triangle-spanning-tree. We describe here a compact implementation of this decompression. A previously proposed alternative, called Spirale Reversi [IsSo99], interprets the reversed *clers* string and builds the triangle tree from the end.

The contributions of this paper are a simple data structure, called the Corner-Table, for representing the connectivity of triangle meshes and a very compact (single page) description of the complete Edgebreaker compression and decompression algorithms, which trivializes their implementation. Because the corner table is nothing more than two arrays of integers and because the decompression is simple and fast, the scheme may be suitable for hardware implementation. We first define our notation and introduce the Corner-Table, then we present the simplified Compression and Decompression algorithms.

Notation and Corner-Table

Vertices are identified using positive integers. Their location is stored in an array called G for "geometry". Each entry of G is a 3D point that encodes the location of a vertex. (Other vertex attributes are ignored for simplicity here. *Sorry*.) We have overloaded the "+" and "–" operators to perform additions and subtraction of points and vectors. Thus G[1] – G[0] returns the vector from the first vertex to the second. Edgebreaker compression stores a point and a sequence of corrective vectors in the string called *delta*, using WRITE(*delta*, D) statements, where D is a point or vector. The corrective vectors will be encoded using a variable length binary format in a separate post-processing entropy-compression step. During decompression, the first call READ(*delta*) returns a decoded version of the first vertex. Subsequent calls to READ(*delta*) return corrective vectors, which are added to the vertex estimates to produce correct vertices.

As pointed out earlier, compression stores, in a string called *clers*, a sequence of symbols from the set {C,L,E,R,S}, encoded using a simple binary format: {0, 110, 111, 101, 100}. We have explicitly used this simple format for completeness, but this code may be easily replaced by a better one. Alternatively, the ASCII symbols may be stored for entropy compression.

During decompression, the symbols (i.e., their binary format) are read and used to switch to the correct operation. We assume that the READ instruction knows to read two more bits when the first one is a 1.

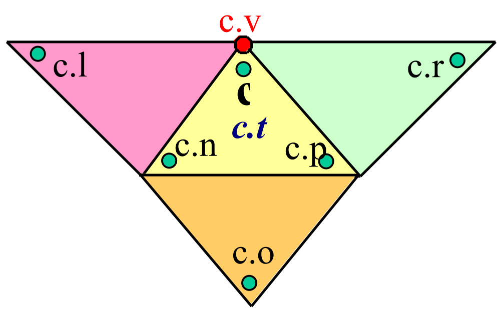
The data structure used by Edgebreaker is composed of two global arrays (the V and O tables) and of two temporary tables (M, U), which are only used during compression. V, O, and U have 3 times as many entries as there are triangles. M has as many entries as vertices. V and O hold the integer references to vertices and to opposite corners. M and U hold binary flags indicating whether the corresponding vertex or triangle has already been visited.

Although Edgebreaker manipulates integer indices, we use (*our own*) object-oriented notation to increase the readability of the algorithms that follow. We use lower-case letters that follow a period to refer to table entries or functions with the corresponding uppercase name. For example, if c is an integer, c.v stands for V[c] and c.o stands for O[c]. However, when we assign values to specific entries in these tables, we still write V[c]=b, rather than c.v=b, to remind the reader that we are updating an entry in the V table. We use left-to-right expansion of this "object-oriented" notation, thus c.o.v stands for V[O[c]].

We also introduce the "next corner around triangle" functions: N(c), which will be written c.n and which returns c–2, if c MOD 3 is 2, and c+1 otherwise. This functions permits to move from one corner of a triangle to the next according to the agreed-upon orientation of the triangle, which we assume to be consistent throughout the mesh. The "previous corner around triangle" function, written as c.p stands for N(N(c)). For example, the statement V[a.p]=b.n.v translates to V[N(N(a))]=V[N(b)].

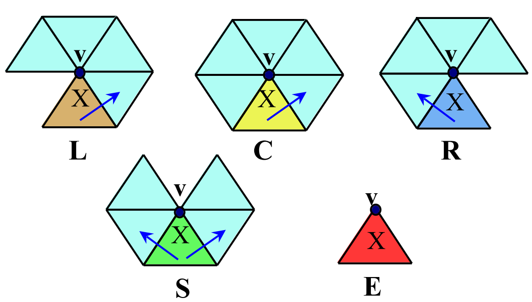
A **corner** c is the association of a triangle c.t with one of its bounding vertices c.v. The entries in V and O are consecutive for the 3 corners (c.p, c, c.n) of each triangle. Thus, c.t returns the integer division of c by 3 and the corner-triangle relation needs not be stored explicitly. For example, when c is 4, c.t is 1 and thus c is a corner of the second triangle. We use c.t only to mark previously visited triangles in the U table.

The notation c.v returns the id of the vertex associated with corner c. We use this id to mark previously visited vertices in the M table or to access the geometry of the vertex (c.v.g). The notation c.o returns the id of the corner opposite to c. To be precise, c.o is the only integer b for which: c.n.v == b.p.v and c.p.v == b.n.v. For convenience, we also define c.l as c.p.o and c.r as c.n.o. These relations are illustrated in the figure below. We assume a counter-clockwise orientation.



Compression

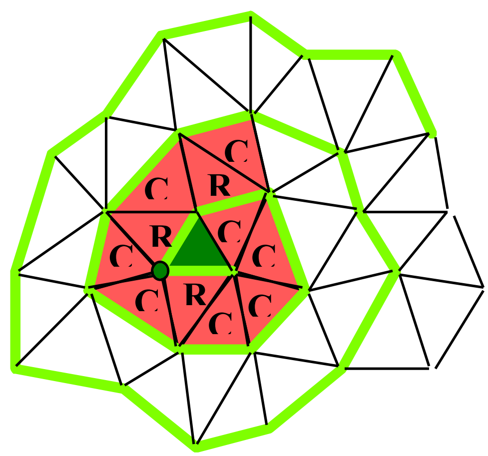
Edgebreaker is a state machine. At each state it moves from a triangle Y to an adjacent triangle X. It marks all visited triangles and their bounding vertices. Let Left and Right denote the other two triangles that are incident upon X. Let **v** be the vertex common to X, Left, and Right. If **v** has not yet been visited, then neither have Left and Right. This is case C. If **v** has been visited, we distinguish four other cases, which corresponds to four situations where one, both, or neither of the Left and Right triangles have been visited. These situations and the associated *clers* symbols are shown in the figure below. The arrow indicates the direction to the next triangle. Previously visited triangles are not shown. Note that in the S case, Edgebreaker moves to the right, using a recursive call, and then to the left.



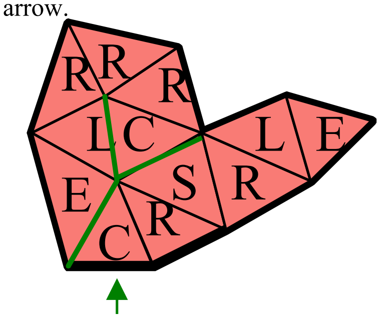
The compression algorithm is composed of an initialization followed by a call to Compress. The initial corner c may be chosen randomly. The initialization decodes and marks the three vertices of the first triangle, marks the triangle as visited, and calls compress.

Compress is a recursive procedure that traverses the mesh along a spiraling triangle-spanning-tree. The recursion starts only at triangles that are of type S and compresses the branch adjacent to the right edge of such a triangle. When the corresponding E triangle is reached, the branch traversal is complete and we "RETURN" from the recursion to pursue the left branch. The encounter of an E that does not match an S terminates the compression process. If the tip vertex of a new triangles has not yet been visited ("IF c.v.m != 1"), we are on a C triangle and we encode in *delta* the corrective vector for the tip of the current triangle using a parallelogram rule [ToGo98]. We also encode a 0 in the *clers* string to indicate a C triangle. When the tip of the new triangle has been visited, we distinguish four cases, based on the status of the neighboring (left and right) triangles.

The figure below shows the labels for triangles that have been visited during a typical early steps of compression, producing the *clers* string CCCCRCCRCRC.



The figure below shows the final steps of compression for a branch or for the whole mesh. It appends the symbols CRSRLECRRRLE to *clers*. The first triangle is marked by an arrow.



```
PROCEDURE initCompression (c){
    GLOBAL M[]={0…}, U[]={0…};                  # init tables for marking visited vertices and triangles
    WRITE(delta, c.p.v.g);                      # store first vertex as a point
    WRITE(delta, c.v.g – c.p.v.g);              # store second vertex as a difference vector with first
    WRITE(delta, c.n.v.g – c.v.g);              # store third vertex as a difference vector with second
    M[c.v] = 1;  M[c.n.v] = 1; M[c.p.v] = 1;    # mark these 3 vertices
    U[c.t] = 1;                                 # paint the triangle and go to opposite corner
    Compress (c.o); }                           # start the compression process

RECURSIVE PROCEDURE Compress (c) {                          # compressed simple t-meshes
    REPEAT {                                                # start traversal for triangle tree
        U[c.t] = 1;                                         # mark the triangle as visited
        IF c.v.m != 1                                       # test whether tip vertex was visited
         THEN {WRITE(delta, c.v.g – c.p.v.g – c.n.v.g + c.o.v.g);  # append correction for c.v
              WRITE(clers, 0);                              # append encoding of C to clers
              M[c.v] = 1;                                   # mark tip vertex as visited
              c = c.r}                                      # continue with the right neighbor
        ELSE IF c.r.t.u == 1                                # test whether right triangle was visited
          THEN IF c.l.t.u == 1                              # test whether left triangle was visited
              THEN {WRITE(clers, 111); RETURN }             # append code for E and pop
              ELSE  {WRITE(clers, 101); c = c.l }           # append code for R, move to left triangle
          ELSE IF c.l.t.u == 1                              # test whether left triangle was visited
              THEN {WRITE(clers, 110); c = c.r }            # append code for L, move to right triangle
              ELSE {WRITE(clers, 100);                      # append code for S
                    Compress(c.r);                          # recursive call to visit right branch first
                    c = c.l } } }                           # move to left triangle
```

Decompression

The decompression algorithm builds the two arrays, V and O, of the corner Corner-Table and also the G table of vertex locations. After initializing the first triangle in *initDecompression*, the recursive procedure *Decompress* is called with corner 1 as parameter. At each iteration of the loop in this procedure, Edgebreaker appends a new triangle to a previously visited triangle. It reads the binary encoding of the next symbol from the *clers* string. If it is a C (binary code 0), Edgebreaker associates the label –1 with the corner opposite the left edge. This temporary marking is stored in the table O. It will be replaced with the correct reference to the opposite corner by a subsequent zip.

If the symbol is an L (binary code 101), Edgebreaker associates a different label (–2) with the opposite edge and tries to zip, by identifying it with the adjacent edge on the left. When an R symbol is encountered, the opposite edge is labeled -2. No zipping takes place. When an E symbol is encountered, both edges are labeled -2, and an iterative zipping is attempted. This zipping will continue as long as the free edge on the right of the last zipped vertex is marked with –2 and the free edge on the left is marked –1. An S symbols forks a recursive call to *Decompress,* which will construct and zip a subset of the mesh that is incident to the right edge of the current triangle. Then the reconstruction proceeds to decode and build the branch attached to the left edge of the current triangle. Typically less than 5% of the triangles are of type S.

Conclusion

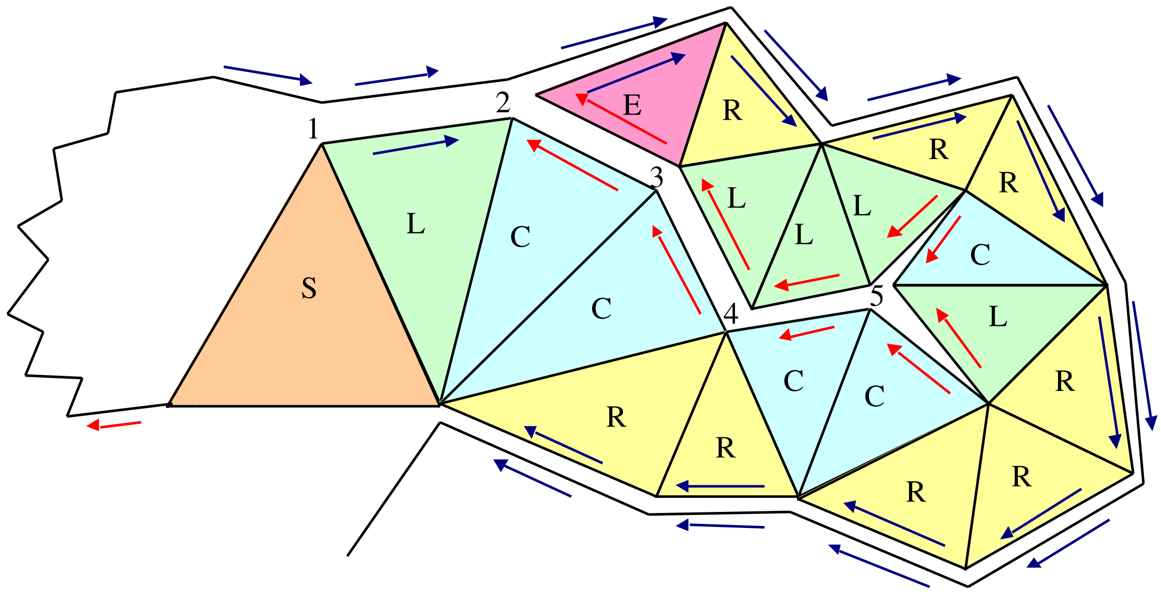
3D mesh compression and planar graph encoding techniques have been the subject of numerous publications (see [Ross99] for a review of prior art). All these approaches have been presented at a high level. Many are complex and difficult to implement. In comparison, the proposed compression and decompression algorithms are trivial to implement. More importantly, the source code is extremely small and uses simple arrays of integers as a data structure. This simplicity makes them suitable for many Internet and possibly even hardware applications.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant 9721358.

```
PROCEDURE initDecompression {
        GLOBAL V[] = { 0,1,2,0,0,0,0,0,…};                    # table of vertex Ids for each corner
        GLOBAL O[] = {–1,–3,–1, –3, –3, –3…};                 # table of opposite corner Ids for each corner
        GLOBAL T = 0;                                         # id of the last triangle decompressed so far
        GLOBAL N = 2;                                         # id of the last vertex encountered
        DecompressConnectivity(1);                            # starts connectivity decompression

        GLOBAL M[]={0…}, U[]={0…};                # init tables for marking visited vertices and triangles
        G[0] = READ(delta);                                   # read first vertex
        G[1] = G[0]+ READ(delta);                             # set second vertex using first plus delta
        G[2] = G[1]+ READ(delta);                             # set third vertex using second plus new delta
        GLOBAL N = 2;                                         # id of the last vertex encountered
        M[0] = 1;  M[1] = 1; M[2] = 1;                        # mark these 3 vertices
        U[0] = 1;                                             # paint the triangle and go to opposite corner
        DecompressVertices(O[1]); }                           # starts vertices decompression

RECURSIVE PROCEDURE DecompressConnectivity(c) {
        REPEAT {                                              # Loop builds triangle tree and zips it up
                T++;                                          # new triangle
                O[c] = 3T; O[3T] = c;                         # attach new triangle, link opposite corners
                V[3T+1] = c.p.v; V[3T+2] = c.n.v;             # enter vertex Ids for shared vertices
                c = c.o.n;                                    # move corner to new triangle
                Switch READ(clers) {                          # select operation based on next symbol
                        Case 0: {O[c.n] =  –1; V[3T] = ++N;}  # C: left edge is free, store ref to new vertex
                        Case 110: { O[c.n] =  –2; zip(c.n); } # L: orient free edge, try to zip once
                        Case 101: { O[c] =  –2; c = c.n }     # R: orient free edge, go left
                        Case 100: { DecompressConnectivity(c);  c = c.n }     # S: recursion going right, then go left
                        Case 111: { O[c] =  –2; O[c.n] =  –2; zip(c.n);  RETURN }}}} # E: zip, try more, pop

RECURSIVE PROCEDURE Zip(c) {                                  # tries to zip free edges opposite c
        b = c.n; WHILE b.o>=0 DO b=b.o.n;                     # search clockwise for free edge
        IF b.o != –1 THEN RETURN;                             # pop if no zip possible
        O[c]=b; O[b]=c;                                       # link opposite corners
        a = c.p;  V[a.p] = b.p.v;                             # assign co-incident corners
        WHILE a.o>=0 && b!=a DO {a=a.o.p; V[a.p]=b.p.v};
        c = c.p; WHILE c.o >= 0 && c!= b DO c = c.o.p;        # find corner of next free edge on right
        IF c.o == –2 THEN Zip(c) }                            # try to zip again

RECURSIVE PROCEDURE DecompressVertices(c) {
        REPEAT {                                              # start traversal for triangle tree
                U[c.t] = 1;                                   # mark the triangle as visited
                IF c.v.m != 1                                 # test whether tip vertex was visited
                  THEN {G[++N] = c.p.v.g+c.n.v.g-c.o.v.g+READ(delta);   # update new vertex
                        M[c.v] = 1;                           # mark tip vertex as visited
                        c = c.r;}                             # continue with the right neighbor
                ELSE IF c.r.t.u == 1                          # test whether right triangle was visited
                   THEN IF c.l.t.u == 1                       # test whether left triangle was visited
                        THEN  RETURN                          # pop
                        ELSE  { c = c.l }                     # move to left triangle
                   ELSE IF c.l.t.u == 1                       # test whether left triangle was visited
                        THEN { c = c.r }                      # move to right triangle
                        ELSE { DecompressVertices (c.r);      # recursive call to visit right branch first
                                c = c.l } } }                 # move to left triangle
```

The *clers* string SLCCRRCCRRRLCRRLLLRE will generate the mesh below. The left edge of the first L triangle is not zipped immediately. The left edge of the second L triangle is zipped reaching vertex 5. Then, as we encounter the subsequent three L triangles, their left edges are zipped right away. The first let edge of the E triangle is also zipped. The rest will be zipped later, when the left branch of the split is done.



REFERENCES

[GuSt98] S. Gumhold and W. Strasser, "Real Time Compression of Triangle Mesh Connectivity", Proc. ACM Siggraph, pp. 133-140, July 1998.

[**IsSo99**] M. Isenburg and J. Snoeyink, "Spirale Reversi: Reverse decoding of the Edgebreaker encoding", Tech. Report TR-99-08, Computer Science, UBC, 1999.

[**King99**] D. King and J. Rossignac, "Guaranteed 3.67V bit encoding of planar triangle graphs", 11th Canadian Conference on Computational Geometry (CCCG''99), pp. 146-149, Vancouver, CA, August 15-18, 1999.

[**King99b**] D. King and J. Rossignac, "Connectivity Compression for Irregular Quadrilateral Meshes" Research Report GIT-GVU- 99 -29, Dec 1999.

[RoCa99] J. Rossignac and D. Cardoze, "Matchmaker: Manifold Breps for non-manifold r-sets", Proceedings of the ACM Symposium on Solid Modeling, pp. 31-41, June 1999.

[**Ross99**] J. Rossignac, "Edgebreaker: Connectivity compression for triangle meshes", IEEE Transactions on Visualization and Computer Graphics, 5(1), 47-61, Jan-Mar 1999. *(Sigma Xi award: Best Paper from Georgia Tech.)*

[**RoSz99**] J. Rossignac and A. Szymczak, "Wrap&Zip decompression of the connectivity of triangle meshes compressed with Edgebreaker", Computational Geometry, Theory and Applications, 14(1/3), 119-135, November 1999.

[**SKR00**] A. Szymczak,. D. King, J. Rossignac, "An Edgebreaker-based efficient compression scheme for regular meshes", Proc of the 12th Canadian Conference on Computational Geometry, Fredericton, New Brunswick, August 16-19, 2000.

[**SKR00b**] A. Szymczak, D. King, J. Rossignac, "An Edgebreaker-based Efficient Compression Scheme for Connectivity of Regular Meshes", Journal of Computational Geometry: Theory and Applications, 2000.

[**TaRo98**] G. Taubin and J. Rossignac, "Geometric Compression through Topological Surgery", ACM Transactions on Graphics, 17(2), 84-115, April 1998. *(IBM award: Best Computer Science Paper from IBM.)*

[ToGo98] C. Touma and C. Gotsman, "Triangle Mesh Compression", Proceedings Graphics Interface 98, pp. 26-34, 1998.