

Edge breaker on a Corner table:

A simple technique for representing and compressing triangulated surfaces

Jarek Rossignol, Alla Safonova, Andrzej Szymczak
College of Computing and GVC Center
Georgia Institute of Technology

Abstract

A triangulated surface S with V vertices is sometimes stored as a list of T independent floating-point coordinates of its vertices. This representation requires about 576 information regarding the adjacency between neighboring triangles or vertices. A vertex structures may be derived from such a representation in order to make explicit the relations between triangles, edges, and vertices. These relations are stored to accelerate in a systematic manner and access the neighbors of each vertex or triangle. Instead of advocate a simple Corner Table, which explicitly represents the triangle/vertex incidence adjacency of any manifold or pseudo-manifold triangle mesh, as two tables of integers. $12V \log_2 2$ bits and must be accompanied by a vertex table, which requires $96V$ bits, if F may be derived from the list of independent triangles. For meshes homeomorphic to a sphere that $4V$ bits by storing the sequence of triangle-labels, if for the further compression to may be guaranteed by using context-based codes for the clers symbols. Entropy codes reduce to less than $2V$ bits. Meshes with more complex topology lead to additional bits per handle or present here a publicly available, simple, state-machine implementation of the Edgebreaker: the corner table, computes the CLERS symbols, and constructs an ordered list of vertices in the order in which they appear on the list, as corrective displacements between them. Quantizing vertex coordinates to 12 bits and predicting each vertex as a linear combination of its neighbors leads to short displacements, for which entropy codes drop the total vertex list typical meshes below $16V$ bits.

Introduction

3D graphics plays an increasingly important role in applications where 3D models are a tool for improved design and model acquisition tools, to the wider acceptance of this technology. Consequently, the number and complexity of these models are growing more rapidly than predicted. Consequently, it is imperative to continue increasing the terseness of 3D data transmission and reliability of the associated compression and decompression algorithms.

Although many representations have been proposed for 3D models, polygon and triangle meshes are the most common for exchanging and viewing 3D models. A triangle mesh may be represented by its vertices and triangles. Vertex data comprises coordinates of all the vertices and optionally the vertex colors and normal vectors and textures. In its simplest form, connectivity captures the incidence of triangles in the mesh and their bounding vertices. It may be represented by a triangle-vertex incidence table where each entry is a triangle the references to its three bounding vertices.

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

Vertex coordinates may be compressed through various forms of vector quantization. Most schemes exploit the coherence in vertex locations by using local or global predictors to encode 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 correction, where shorter the codes. The decoder receives the correction, decodes it and adds it to the predicted vertex information for the next vertex. Thus the prediction can only exploit data that has been transmitted before. Most predictive schemes require only local connectivity between the next vertex and its neighbors.

Some global predictors require having the connectivity of the entire mesh. Thus it is compression techniques that are independent of vertex data.

The Edgebreaker compression scheme, discussed here, has been extended to manifold meshes [Ross99], to triangulated boundaries of non-manifold solids meshes that contain only quad combination of simply-connected polygonal faces with an arbitrary number of sides [King99]. Nevertheless, for sake of simplicity, in this our focus to meshes that are each homeomorphic to a sphere.

As several other compression schemes [TaRo98, Edgebreaker98] visits the triangles in a first) triangle-spanning-tree order and generates a string of descriptors, one per triangle rebuilt by attaching new triangles to previously reconstructed ones. The popularity descriptors are symbols from the set {C,L,E,R,S}. No other parameter is needed. Because trivial code (C=0, L=110, E=111, R=101, S=100) guarantees that storage will not exceed more complex code guarantees per triangle [King99]. This upper bound on storage does not entropy or arithmetic coding schemes, which in general perform poorly on small or irregular simple encoding of Edgebreaker is particularly attractive for large catalogs of small meshes. For large meshes, entropy codes may be used to further reduce the storage [RoSz99]. The string of descriptors produced by Edgebreaker is an efficient decomposition algorithm for the sequence [RoSz99] interprets the symbols to build a simply connected triangle represents the triangle-spanning tree. Then, it zips up the borders of that polygon by in a bottom-up order with respect to the vertex-spanning-tree that is the dual of the here a compact implementation of this decompression. A previously proposed [RoSz99] interprets the reversed and builds the triangle tree from the leaves.

The contributions of this chapter are a simple data structure, called the Corner-Table triangle meshes and a very compact (single page) description of the complete Edgebreaker algorithms, which trivializes their implementation. The data structure, examples, and code are publicly available. Because the corner table is nothing more than two arrays of decompression is simple and fast, the scheme may be suitable for hardware implementation and introduce the Corner-Table, then we present the simplified Compression and Decompression

Notation and Corner-Table

Vertices are identified using positive integers. Their locations are stored in an array G is a 3D point that encodes the location of a vertex. (Other vertex attributes are ignored have overloaded the + and - operators to perform additions and subtractions. Glib-Gid() returns the vector from the first vertex to the second. Edgebreaker compression stores vectors in the string by WRITE(dD) statements, where D is a point or vector. The will be encoded using a variable length binary format in a separate post-processing decompression, the READ(dD) returns a decoded version of the first vertex. Subsequent return corrective vectors, which are added to the vertex estimates to produce correct vertex Compression stores, in a string a sequence of symbols from {C,L,E,R,S}, {encoded using a binary format: {0, 110, 111, 101, 100}. Better codes may be substituted easily, if desired. During decompression, the symbols (i.e., their binary format) are read and used to assume that the READ instruction knows to read two more bits when the first one is a 1.

The Corner Table data structure used by Edgebreaker is composed of two global arrays temporary tables (M, U), which are only used during compression. V, O, and U have 3 triangles. M has as many entries as vertices. V and O hold the integer references to vertices. U hold binary flags indicating whether the corresponding vertex or triangle has already been visited. Although Edgebreaker manipulates integer indices, object-oriented notation to increase the readability of the algorithms that follow. We use lower-case letters that follow a period to refer to the corresponding uppercase name. For example, if c is an integer, c.v stands for V[c] and we assign values to specific entries in these tables, we still write vind=the reader then updating an entry in the V table. We use left-to-right expansion of this so.vj and v[O[c]].

We also introduce the next corner around triangle functions c.M(c) and which returns c-1 MOD 3 is 2, and c+1 otherwise. This functions permits to move from one corner of a triangle agreed-upon orientation of the triangle, which we assume to be consistent throughout

around triangle function, written as $V[N(N(a))]=V[N(b)]$. For example, the statement $V[N(N(a))]=V[N(b)]$ 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 $c.v$ (see Fig.1). 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.

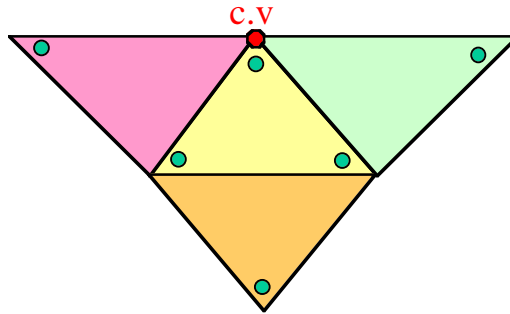


Fig 1 Using the V and O tables, given a corner, c , we can access vertex, the adjacent and next corners, $c.p$, and $c.n$; the opposite corner, $c.o$; and the corners of the left and right triangles, $c.l$ and $c.r$.

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 Fig. 2. 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.

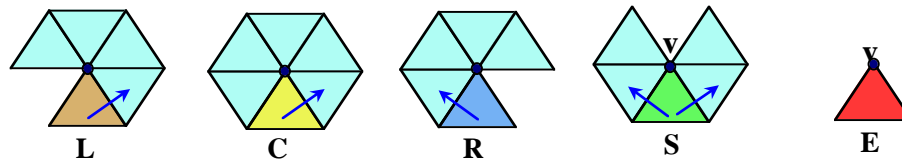


Fig 2 Using the V and O tables, given a corner, c , we can access vertex, the adjacent and next corners, $c.p$, and $c.n$; the opposite corner, $c.o$; and the corners of the left and right triangles, $c.l$ and $c.r$.

The compression algorithm is composed of an initialization followed by a call to *Compress*. The initial corner c may be chosen randomly. The initialization encodes 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-triangles that are of type S . It compresses the branch adjacent to the right edge of S . If the right edge of S is reached, the branch traversal is complete and we **RETURN** from the recursive encounter of an E that does not match an S terminates the compression process. If the vertex v has not yet been visited (**IF** $c.v$, we are on a C triangle and we encode it in the vector for the current triangle using a parallel bit stream. We also encode a C symbol (for example $code(C)$ marks the c string). When the tip of the new triangle has been visited, we distinguish among the status of the neighboring (left and right) triangles. The compression pseudo-code is

```

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, code(C));           # 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, code(E)); RETURN } # append code for E and pop
    ELSE { WRITE(clers, code(R)); 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, code(L)); c = c.r } # append code for L, move to right triangle
    ELSE { WRITE(clers, code(S));          # append code for S
           Compress(c.r);                 # recursive call to visit right branch first
           c = c.l } } }                   # move to left triangle

```

The Fig.3, below, shows the labels for triangles that have been visited during a typical

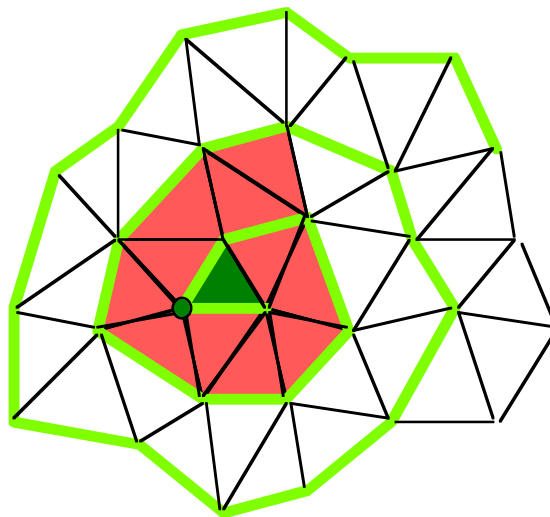


Fig 3 The *clers* sequence CCCCRCRC is produced as Edgebreaker starts compressing this

Fig.4 shows the final steps of compression for a branch or for the whole mesh. It appen to *clers*. The first triangle is marked by an arrow.

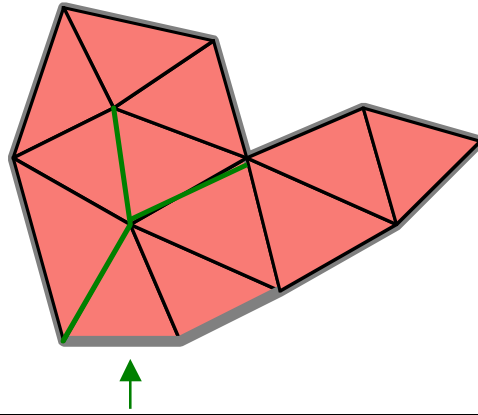


Fig 4 Thelers sequence CRSRLECRRLLE is produced as Edgebreaker finishes compressing the

Decompression

The decompression algorithm builds the two arrays, V and O , of the corner Corner-Table and the G table of vertex locations. After initializing the first triangle in *initDecompression*, the recursive procedure *Decompress* is called for corner 1. In each iteration, Edgebreaker appends a new triangle to a previously visited one. It interprets the binary encoding of the next symbol from the *clers* string. If it is a C , Edgebreaker associates the label -1 with the corner opposite the left edge and stores it in O . This temporary marking will be replaced with the correct reference to the opposite corner by a zip. If the symbol is an L , 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 symbol 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 about 2% of the triangles are of type S . The process is illustrated in Fig. 5.

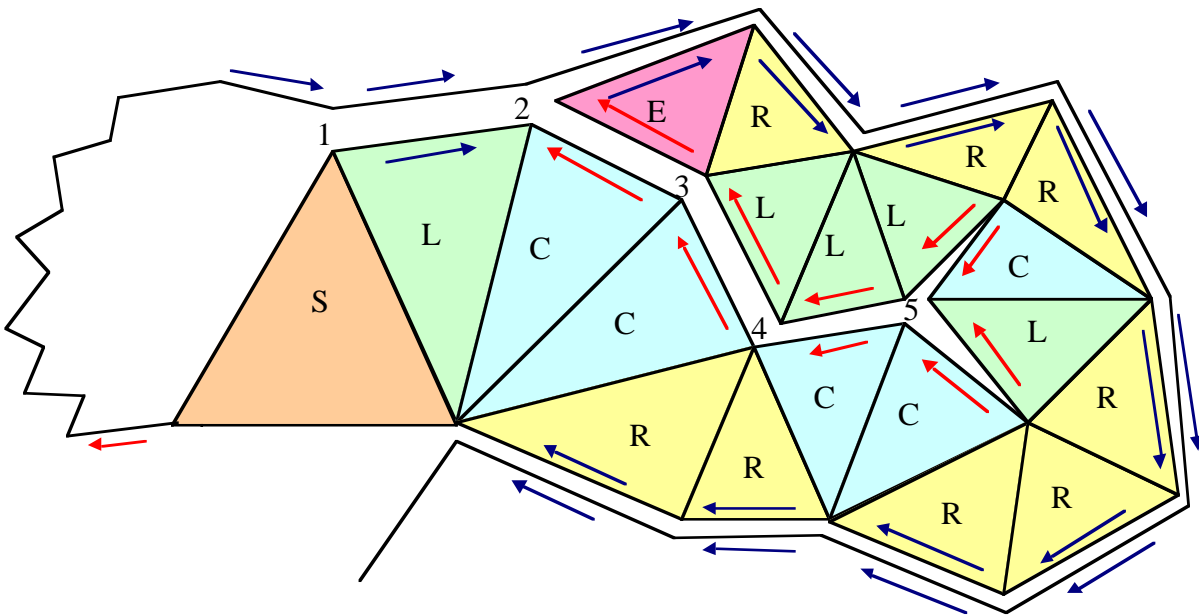


Fig 5: The *clers* string SLCCRRCRRRLCRLLLLRE will generate the mesh shown here. The first L triangle is not zipped immediately. The left edge of the second L triangle. Then, as we encounter the subsequent three L triangles, their left edges are zipped of the E triangle is also zipped. The rest will be zipped later, when the left branch

```

PROCEDURE initDecompression {
    GLOBAL V[] = { 0,1,2,0,0,0,0,...};           # 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 decode(READ(clers)) {          # select operation based on next symbol
            Case C: { O[c.n] = -1; V[3T] = ++N; } # C: left edge is free, store ref to new vertex
            Case L: { O[c.n] = -2; zip(c.n); } # L: orient free edge, try to zip once
            Case R: { O[c] = -2; c = c.n } # R: orient free edge, go left
            Case S: { DecompressConnectivity(c); c = c.n } # S: recursion going right, then go left
            Case E: { 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 pseudocode for the decompression algorithm is shown in the frame above.

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.

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.