# 3D mesh compression

**Jarek Rossignac**

College of Computing and GVU Center
Georgia institute of Technology

In this chapter, we discuss 3D compression techniques for reducing the delays in transmitting triangle meshes over the Internet. We first explain how vertex **coordinates**, which represent surface samples may be compressed through quantization, prediction, and entropy coding. We then describe how the **connectivity**, which specifies how the surface interpolates these samples, may be compressed by compactly encoding the parameters of a connectivity-graph construction process and by transmitting the vertices in the order in which they are encountered by this process. The storage of triangle meshes compressed with these techniques is usually reduced to about a **byte per triangle**. When the exact geometry and connectivity of the mesh are not essential, the triangulated surface may be simplified or **retiled**. Although simplification techniques and the progressive transmission of refinements may be used as a compression tool, we focus on recently proposed retiling techniques designed specifically to improve 3D compression. They are often able to reduce the total storage, which combines coordinates and connectivity, to half-a-bit per triangle without exceeding a mean square error of 1/10,000 of the diagonal of a box that contains the solid.

## BACKGROUND AND TERMINOLOGY

A triangle mesh is defined by a set of **vertices** and by its triangle-vertex **incidence** graph. The vertex description comprises **geometry** (3 coordinates per vertex) and optionally **photometry** (surface normals, vertex colors, or texture coordinates), which will not be discussed here (see [IsSn00, GaHe98, BPZ99] for more information). **Incidence** (sometimes referred to as "topology") defines each triangle by the 3 integer indices that identify its vertices. For simplicity and elegance, we restrict our discussion in this chapter on **simple meshes**, which are homeomorphic to a triangulation of a sphere. However, most of the techniques presented here work for, or have been extended to, more general meshes with borders, handles, non-manifold degeneracies, and non-triangular faces [Ross99, GBTS99, RoCa99, ToGo98, RSS01, KRS99, BPZ99, IsSn00, Lo&02] and even to tetrahedral meshes [SzRo99, SzRo00, PRS99].

In what follows, we assume that our triangle mesh is a connected manifold surface with no boundary and no handle and that it has $v$ vertices, $e$ edges, and $t$ triangles. To simplify the formalism, we consider the **edges** to be the relatively open line segments that do not include their endpoints. Similarly, we use the term **face** to denote the relative interior of a triangle, excluding its edges and vertices. The **surface** of the mesh is the point-set union of its faces, edges, and vertices, which are all pair-wise disjoint.

In order to prove the linear equation linking $t$ and $v$ and to facilitate the description of several compression approaches, we will use the following terminology. A **Vertex-Spanning Tree** (VST) of a triangle mesh is a subset of its edges, selected so that their union with all the vertices forms a tree (connected cycle-free graph). Consider that a given VST has been selected. The edges it contains are called the **cut-edges**. The union of the cut-edges with all the vertices is called a **cut**. Because the VST is a tree, **there are $v$–1 cut-edges**. The difference between the surface and its cut is called the **web**. Edges that are not cut-edges are called **hinge-edges**. The web is composed of all the faces and of all the hinge-edges. Removing the cut, which has no loop, from the surface of a mesh will not disconnect it and will produce a web that is a (relatively open) triangulated two-dimensional point-set in three-space. Because by definition a simple mesh has no hole or handle, the web is simply connected and may be represented by an acyclic graph, whose nodes correspond to faces and whose links correspond to hinge edges. Thus **there are $t$–1 hinge edges**. Note that by picking a leaf of this graph as root and orienting the links, we can always turn it into a binary tree, which we call the **Triangle-Spanning-Tree** (TST). It is a spanning tree of the dual of the graph made of the edges and vertices of the mesh. The TST defines a connected network of corridors through which one may visit all the triangles by walking across hinge-edges and never crossing a cut-edge. Because an edge is either hinge or cut, the **total number of edges**, $e$, **is $v$–1+$t$–1**. Each triangle uses 3 edges and each edge is used by 2 triangles. Thus the number $e$ of edges is also equal to $3t/2$. Combining these two equations yields $t$=**2$v$–4**, which shows that there are roughly twice as many triangles as vertices.

When 32-bit integers are used to represent triangle-vertex incidence references and 32-bit floats to represent vertex coordinates, an **uncompressed representation** of a simple mesh requires $12v$ bytes to store the geometry and $12t$ bytes (or equivalently $24v$–28 bytes) to store the incidence, which amounts to a total of **144$t$ bits**. Note that, surprisingly, the incidence information requires twice more storage than the geometry.

# CORNER TABLE REPRESENTATION

The Corner Table [RSS01, RSS02] is a simple data structure which simplifies the storage and processing of manifold triangle meshes, whether they are simple or have holes and handles. We introduce it here and use it in this chapter to clarify the implementation details of compression and decompression techniques.

The **geometry** is stored in the **coordinate table**, **G**, where G[v] contains the triplet of the coordinates of vertex number v, and will be denoted v.g. Note that the order in which the vertices are listed in G is arbitrary, although once it is chosen, it defines the integer reference number associated with each vertex.

Triangle-vertex **incidence** defines each triangle by the three integer references to its vertices. These references are stored as **consecutive** integer entries in the **V table**. Note that each one of the 3*t* entries in V represents a **corner** (association of a triangle with one of its vertices). Let c be such a corner. Let c.t denote its triangle and c.v its vertex. Remember that c.v and c.t are integers in [0,*v*–1] and [0,*t*–1] respectively. Let c.p and c.n refer to the **previous** and **next** corner in the cyclic order of vertices around c.t.

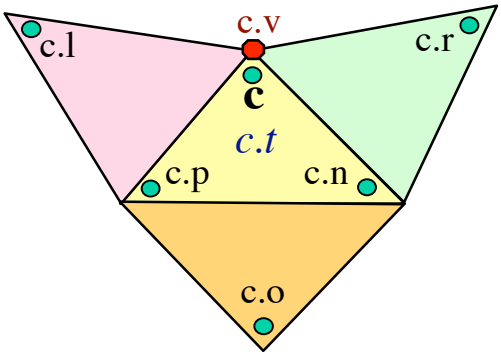


**FIGURE 1:** *Corner operators for traversing a corner table representation of a triangle mesh.*

Although G and V suffice to completely specify the triangles and thus the surface they represent, they do not offer direct access to a neighboring triangle or vertex. We chose to use the reference to the **opposite** corner, c.o, which we cache in the **O table** to accelerate mesh traversal from one triangle to its neighbors. For convenience, we also introduce the operators c.l and c.r, which return the **left** and **right neighbors** of c (Fig. 1).

Note that we do not need to cache c.t, c.n, c.p, c.l, or c.r, because they may be quickly evaluated as follows: c.t is the integer division c.t DIV 3; c.n is c–2, when c MOD 3 is 2, and c+1 otherwise; and c.p is c.n.n; c.l is c.n.o; and c.r is c.p.o. Thus, the storage of the connectivity is reduced to the O and V arrays.

We assume that all triangles have been consistently **oriented**, so that c.n.v=c.o.p.v for all corners c. For example, one may adhere to the convention that when a triangle c.t is visible by a viewer outside of the solid (i.e., the finite set that is bounded by the triangle mesh), the three vertices, c.p.v, c.v, and c.n.v, appear in clockwise order.

Assume that c.t.m is a Boolean set to **TRUE** when the triangle c.t has been visited. The procedure **visit**(c) {**IF** NOT c.t.m **THEN** {c.t.m:=TRUE; **visit**(c.r); **visit**(c.l)}} will visit all the triangles in a depth-first order of a TST.

Given the V table, the entries in O may be computed by: **FOR** c:=0 **TO** 3t–2 **DO FOR** b:=c+1 **TO** 3t–1 **DO IF** (c.n.v==b.p.v)&&(c.p.v==b.n.v) **THEN** {c.o:=b; b.o:=c}. A faster approach sorts the triplets {min(c.n.v,c.p.v), max(c.n.v,c.p.v), c} into bins. All entries in a bin have the same first record: min(c.n.v,c.p.v), an integer in [0,v–1]. There are rarely more than 20 entries in a bin. Then, we sort the entries in each bin by the second record: max(c.n.v,c.p.v). Now, pairs with identical first record and with identical second record are consecutive and correspond to opposite corners, which are identified by the third record in each triplet. Thus, if a sorted bin contains consecutive entries (a,b,c) and (a,b,d), we set c.o:=d and d.o:= c.

Because it can be easily recreated, the O table needs not be transmitted. Furthermore, the $31-\log_2 v$ leading zeros of each entry in the V table need not be transmitted. Thus, assuming that Floats are used for the coordinates, a compact, but **uncompressed** representation of a triangle mesh requires **48*t* bits** for the coordinates and $\mathbf{3}t\log_2 v$ **bits** for the V table. Note that Edgebreaker (discussed below) encodes the full connectivity information contained in both V and O with a linear cost of less than 2*t* bits, and hence eliminates the need for the decompression modules on the client to re-compute O from V.

# GEOMETRY COMPRESSION

The compression of vertex coordinates usually combines three steps: quantization, prediction, and statistical coding of the residues. We explain them briefly in this section.

**Quantization** truncates the vertex coordinates to a desired accuracy and maps them into integers that can be represented with a limited number of bits. To do this, we first compute a tight (min-max), axis-aligned bounding box around each object. The minima and maxima of the x, y, and z coordinates, which define the box, will be encoded and transmitted with the compressed representation of each object. Then, given a desired accuracy, e, we transform each x coordinate into an integer $i=\mathrm{INT}((x-x_{min})/(e(x_{max}-x_{min})))$, which ranges between 0 and $2^B$, where $B=\lceil\log_2((x_{max}-x_{min})/e)\rceil$ is the maximum number of bits needed to represent the quantized coordinate i. The

y and z coordinates are quantized similarly. Choosing $e=\max((x_{max}-x_{min})/2^{12}, (y_{max}-y_{min})/2^{12}, (z_{max}-z_{min})/2^{12})$ yields $B\leq 12$ for all coordinates and ensures a sufficient geometric fidelity for most applications and most models. Thus, this **lossy** quantization step reduces the storage cost of geometry from $96v$ bits to less than **$36v$ bits**.

The next, and most crucial, geometry compression step involves using a **vertex predictor**. Both the encoder and the decoder use the same predictor. Thus, only the **residues** between the predicted and the correct coordinates need to be transmitted. The **coherence** between neighboring vertices in meshes of finely tiled smooth surfaces reduces the magnitude of the residues.

Because most edges are short with respect to the size of the model, adjacent vertices are in general close to each other and the differences between the coordinates are small. Thus a new vertex may be predicted by a previously transmitted **neighbor** [Deer95].

Instead of using a single neighbor, when vertices are transmitted in VST top-down order, a **linear combination** of the 4 **ancestors** in the VST may be used [TaRo98]. The 4 coefficients of this combination are computed to minimize the magnitude of the residues over the entire mesh and transmitted as part of the compressed stream.

The most popular predictor for single-rate compression is based on the **parallelogram** construction [ToGo98]. Assume that the vertices of c.t have been decoded. We predict c.o.v.g using c.n.v.g+c.p.v.g–c.v.g. The parallelogram prediction may sometimes be improved by predicting the angle between c.t and c.o.t from the angles of previously encountered triangles or from the statistics of the mesh.

Some of the residues may be large. Thus, good prediction, by itself may not lead to compression. For example, if the coordinates have been quantized to B-bit integers, some of the coordinates of the corrective vector, c.o.v.g –c.n.v.g –c.p.v.g +c.v.g may require B+2 bits of storage. Thus, parallelogram prediction could in principle expand storage rather than compress it. However, the distribution of the residues is usually biased towards zero, which makes them suitable for statistical compression [Salo00].

In practice, the combination of these steps compresses vertex location data to about **$7t$ bits**.

# CONNECTIVITY COMPRESSION

As argued above, geometry may be encoded efficiently, provided that connectivity information is available during geometry decompression to locate previously decoded neighbors of each vertex. This section presents techniques for compressing the connectivity information from $3t\log_2 v$ bits to $bt$ bits, where $b$ is guaranteed never to exceed 1.80, and in practice is usually close to 1.0. As a result, meshes may be encoded with a total of about **$8t$ bits** ($7t$ bits for geometry, $1t$ bit for connectivity).

Instead of retracing the chronological evolution of the research in the field of single-rate incidence compression, we first describe in detail Edgebreaker [Ross99], which is arguably the simplest and one of the most effective single-rate compression approaches. The source code for Edgebreaker is publicly available [SaRo02]. Then, we briefly review several variants and other approaches, using Edgebreaker's terminology to characterize their main differences and respective advantages or drawbacks.

## Edgebreaker

The Edgebreaker **compression** visits the triangles in a spiraling (depth-first) TST order and generates the ***clers* string** of labels, one label per triangle, which indicate to the decompression how the mesh can be rebuilt by attaching new triangles to previously reconstructed ones (Fig. 2).

```
RECURSIVE PROCEDURE Compress (c) {                # compresses a simple t-meshes
        REPEAT {                                  # traverses TST, stopped by RETURN
          set(c.t.m);                             # mark triangle as visited
          IF !c.v.m                               # test whether tip vertex was visited
            THEN { encode(c.v.g);                 # store location of tip
              WRITE(clers, 'C');                  # append C to clers string
              set(c.v.m);                         # mark tip vertex as visited
              c:= c.r}                            # continue with the right neighbor
            ELSE IF c.r.t.u                       # test whether right triangle was visited
            THEN IF c.l.t.u                       # test whether left triangle was visited
              THEN {WRITE(clers, 'E'); RETURN }   # append code for E and pop or stop
              ELSE  {WRITE(clers, 'R'); c:= c.l } # append code for R, move to left triangle
            ELSE IF c.l.t.u                       # test whether left triangle was visited
              THEN {WRITE(clers, 'L'); c:= c.r }  # append code for L, move to right triangle
              ELSE {WRITE(clers, 'S');            # append code for S
                Compress(c.r); c:= c.l }} }       # recurse right, then continue left
```

The Edgebreaker compression pseudo-code is shown in the insert above. The following explanations contain in parentheses the excerpts of the pseudo-code they reference. Edgebreaker works directly on the Corner Table and does not require any additional data structure, except for one bit per vertex and one bit per triangle to mark the ones that have already been processed. In particular, it does not require maintaining linked lists of border edges. It traverses the mesh in depth-first order of a TST using iteration (REPEAT) and occasionally recursion (*Compress*) on corner indices. It marks all visited vertices (set(c.v.m)) and triangles (set(c.t.m)). The current triangle is identified by its tip corner (c). Note that the current triangle has been reached though the **gate** edge joining c.n.v with c.p.v. By testing the marks of the tip vertex of the current triangle and of neighboring triangles, it selects the label and appends it to the sequence the *clers* string.

If the tip vertex (c.v) has not yet been visited (!c.v.m), its location is encoded (encode(c.v.g)) using the parallelogram prediction and geometry compression, as explained earlier. The label C is appended to the *clers* string (WRITE(clers, C)) and the iteration moves to the right neighbor (c:=c.r). Note that the vertices are encoded in the **order** in which they are encountered by C-triangles during this traversal. This order does not usually reflect the order in which the vertices were listed in the original mesh. Similarly, the triangles are reordered during transmission. A dictionary mapping the original order on the server to the new order on the client may be kept on the server to reconcile vertex or triangle selections between one location and the other in subsequent processing.

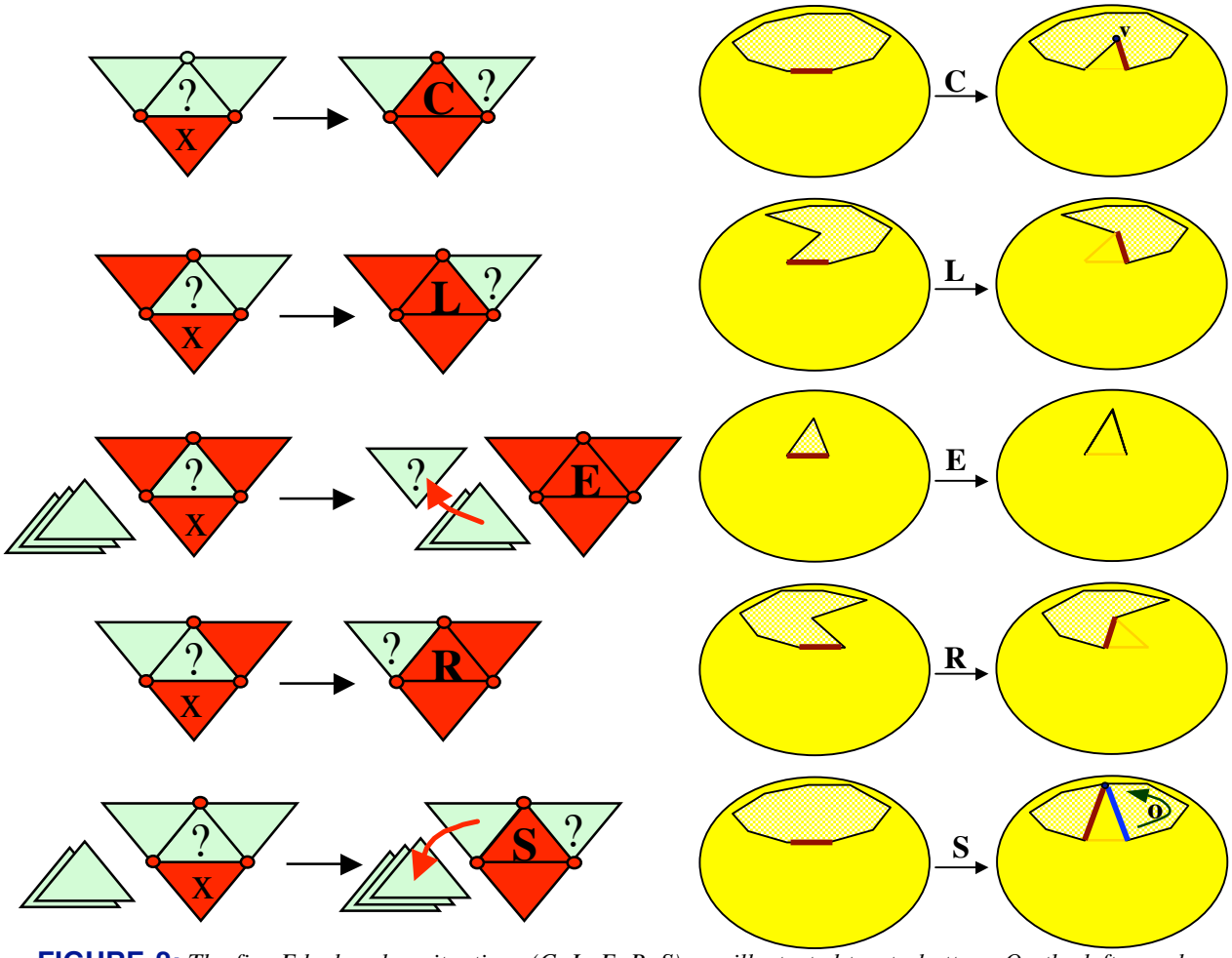


**FIGURE 2:** *The five Edgebreaker situations (C, L, E, R, S) are illustrated top-to-bottom. On the left, we show the "before" and "after" states for each situation during* ***compression****. The current triangle is marked by "?". Previously visited triangles and vertices are darker. An X marks the triangle through which we came. We encode a C label when the tip vertex of the current triangle was not marked (top). Otherwise, the label depends on the status of the left and right neighbor triangles. When neither were visited, we encode an S (bottom), after which compression goes right via a recursive call and then left. We show this symbolically by adding the left neighbor to a stack. When both neighbors have been visited previously, we encode and E and exit the procedure (possibly returning from a recursive call). The right column shows how* ***decompression*** *interprets the CLERS symbols to reconstruct the connectivity of the mesh. For each symbol in the clers string, Edgebreaker decompression attaches a new triangle to the gate edge (indicated by a thick line on the left figure, where the state before the insertion of the new triangle is shown). The gate for the next operation is placed as indicated by a thick line on the right column, which shows the state after the new triangle was inserted. Decoding a C symbol (top) creates a new vertex (v). When decoding an S symbol (bottom), the location of the tip of the new triangle is defined by the offset o from the gate around the bounding loop. The S operation puts the gate on the right edge of the new triangle and proceeds to fill the right hole using a recursive call. Then it sets the gate to the left edge of the new triangles and resumes the process. The offsets o for each S symbol may be computed from the clers string using the fact that C and S increment the edge count, L and R decrement it, and E reduces it by 3.*

When the tip of the current triangle has been previously visited, we distinguish four other types of triangles: L, R, S, and E (Fig. 2).

**case L**: When the left neighbor has been visited, but not the right one, we append the label L to the *clers* string and iterate on the right neighbor (c:= c.r).

**case R**: When the triangle on the right has been visited, but not the one on the left, we append R to the *clers* string and iterate on the left neighbor (c:= c.l).

**case S**: When both neighbors have not-visited status, we append S to *clers*, start a recursive process on the right neighbor (***Compress***(c.r)), and then iterate on the left neighbor (c:= c.l).

**case E**: When both neighbors have been visited, we append E to the *clers* string, and return from recursion or from the compression process (**RETURN**).

The connectivity of the first triangle is implicit. The initialization, detailed in the insert below, sets the visited tags (.m) to zero (not shown here). Then it encodes the first three vertices and marks them and their triangle. It calls the compression on one of the corners of that triangle (*Compress* (c.o)).

```
PROCEDURE initCompression (c){
        encode(c.p.v.g); encode(c.v.g); encode(c.n.v.g);   # store first 3 vertices
        set(c.v.m, c.p.v.m, c.n.v.m, c.t.m);               # mark first 3 vertices and triangle as visited
    Compress (c.o); }      # start compression at opposite corner
```

A typical execution of the compression process is illustrated in Fig. 3.

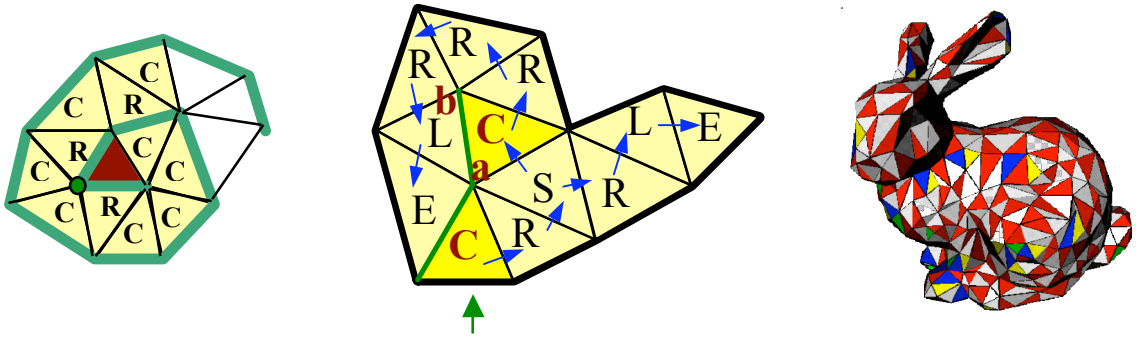


**FIGURE 3:** *In this example of a typical compression situation, Edgebreaker starts with the darker triangle (left) and spirals out clockwise, filling the beginning of the clers string with CCCCRCCRCRC. It appends the tip of each C triangle to the vertex list. A typical situation where Edgebreaker finishes compression or closes a hole is shown (center). It spirals counterclockwise, appending the label sequence CRSRLECRRRLE to the clers string and adding the vertices (a) and (b) to the vertex list. The triangles in the rabbit (right) have been shaded according to their Edgebreaker labels. Notice that half of the triangles are C (white) and about a third are R.*

Because, except for the first two vertices, there is a one-to-one mapping between each C triangle and each vertex, the number of C triangles is $v$–2. Consequently, the number of non-C triangles in a simple mesh is $t$–($v$–2), which is also $v$–2. Thus exactly **half of the triangles are of type C**. Hence, Edgebreaker guarantees that a compressed representation of the connectivity of a simple triangle mesh will never exceed **2*t* bits** [Ross99] if we use the following simple binary code for the labels (C=0, L=110, E=111, R=101, S=100).

Given that the subsequences CE and CL are impossible, a slightly more complex code [KiRo99] may be used to guarantee that the compressed file will never exceed **1.84*t* bits**. This code uses (C=0, S=10, and R=11) for symbols that follow a C and one of the following 3 codes for symbols that do not follow a C:

- **Code 1:** C is 0, S is 100, R is 101, L is 110, E is 111
- **Code 2:** C is 00, S is 111, R is 10, L is 110, E is 01
- **Code 3:** C is 00, S is 010, R is 011, L is 10, E is 11

It was proven [KiRo99] that one of these 3 codes always takes less than 11$t$/6 bits. A 2-bit switch header is used to identify which code is used for each model.

Further constraints exist on the *clers* string. For example, CCRE is impossible, because CCR increments the length of the loop, which must have been at least 3. By exploiting such constraints to better estimate the probability of the next symbol, a more elaborate code was developed [Gumh00], which guarantees 1.778$t$ bits when using a forward decoding [RoSz99] and **1.776*t* bits** when using a reverse decoding scheme [IsSn01].

Hence, the Edgebreaker encoding of the connectivity of any mesh (homeomorphic to a sphere) may be compressed down to 1.78$t$ bits. This brings it within 10% of the proven 1.62$t$ **theoretical lower bound** for encoding planar triangular graphs, as established by [Tutt62], who by counting all planar triangulations of $v$ vertices has proven that an optimal encoding uses at least $v\log_2(256/7)\approx 3.245v$ bits, for a sufficiently large $v$.

These recent developments constitute a significant advance in the study of short encodings of planar triangle graphs. They are often the best solution for compressing small or irregular meshes. For large and fairly regular meshes, better compression ratios may often be obtained. For example, one may encode CC, CS, and CR pairs as single symbols. Each odd C symbol will be paired with the next symbol. After an even number of C symbols, we use the following codes: CR=01, CC=00, CS=1101, R=10, S=1111, L=1110, E=1100. This encoding guaranteed 2.0$t$ bits, but usually yields between 1.3$t$ and 1.6$t$ bits [RoSz99].

Furthermore, by arranging symbols into words that each start with a sequence of consecutive Cs and by using a Huffman code [Salo00], we often reduce storage to less than 1.0$t$ bits. For example, 0.85$t$ bits suffice for the

Huffman codes of the Stanford Bunny. Including the cost of transmitting the associated 173 words dictionary brings the total cost to **0.91*t* bits**. A gzip compression of the resulting bit stream reduces it by only 2%.

As shown earlier, the location of the next vertex may be predicted using previously decoded geometry and connectivity. Coors and Rossignac [CoRo02] have proposed to also **predict** the **connectivity** of the next triangle using the same information. In their **Delphi** system, compression and decompression perform the same **geometric prediction** of the location of the tip-vertex of the next triangle. Then they estimate the triangle connectivity, and thus its symbol in the CLERS string produced by the Edgebreaker compression, by **snapping** the tip-vertex to the nearest vertex in the active loop, if one lies sufficiently close. If no bounding vertex lies nearby, the next *clers* symbol is estimated to be a C. If the guess is correct, a single confirmation bit is sufficient. Otherwise, an entropy-based code is received and used to select the correct CLERS symbol from the other four possible ones (or the correct tip of an S triangle). Reported experiments indicate that, depending on the model, up to 97% of Delphi's guesses are correct, compressing the connectivity down to **0.19*t* bits**. When the probability of a wrong guess exceeds 40%, the Delphi encoding stops being advantageous.

Let us now discuss several approaches to the **decompression** of the *clers* string. All approaches attach a new triangle, one at a time, to a **gate** edge, which so far has only one incident triangle. The next symbol in the *clers* string defines where the tip of the new triangle is (Fig. 2). Symbol C indicates that the new triangle will have as tip a **new** vertex. Note that the three vertices of the previously decoded triangle that is incident upon the gate have been previously decoded and may be used in a parallelogram prediction of the new vertex. Also note that the numbering of the vertices and hence their order in the G table of the reconstructed mesh reflects the order in which the vertices are instantiated as tips of C triangles.

Symbol L indicates that the tip vertex is immediately to the left of the gate along the boundary of the portion of the mesh decoded so far. R indicates that the tip immediately the right of the gate. E indicates that the new triangle will close a hole, which must have exactly 3 vertices. S indicates that the tip of the new triangle is elsewhere on the boundary of the previously decoded portion of the mesh.

Consider the edge-connected components of the not-yet decoded portion of the mesh. Let M be the component incident upon the gate. Because by definition of simple meshes M has no handle, an S triangle will always split it in two parts. Through a recursive call, Edgebreaker will first reconstruct the portion of M that is incident upon the right edge of S, as seen when entering the triangle through the gate. Then, upon return from the recursive call, the reconstruction of the rest of M will resume.

After the new triangle is attached, the gate is moved to the right edge of the new triangle for cases C and L. It is moved to the left edge for case R. When an S triangle is attached, the gate is first moved to the right edge of the S triangle and the right hole is filled through a recursive call to decompression. Then the gate is moved to the left edge and the process resumes as if the S triangle had been an R triangle.

The only challenge in the Edgebreaker decompression lies in the location of the tips of the S triangles. Several approaches have been proposed and are briefly discussed below.

The integer reference number of the tip of each S triangle could be encoded using $\log_2 k$ bits, where $k$ is the number of previously decoded vertices. A more economical approach encodes an **offset**, $o$, indicating the number of vertices that separate the gate from the tip in the current loop (Fig. 2). Because the current loop may include a large fraction of the vertices, one may still need up to $\log_2 k$ bits to encode the offset. Although the total cost of encoding the offsets is linear in the number of triangles [Gumh99], the encoding of the offsets constitutes a significant fraction of the total size of the compressed connectivity. Hence, several authors strived to minimize the number of offsets [AlDe01b], mostly by using heuristics for selecting gates with a low probability of being the base of S triangles.

The breakthrough of Edgebreaker lies in the discovery that **offsets need not be transmitted** at all, because they can be recomputed by the decompression algorithm from the *clers* string itself. The initial solution [Ross99] is based on the observation that the attachment of a triangle of each type changes the number of edges in the current loop by specific amounts (Fig. 2). C increments the edge-count. R and L decrement it. E removes a loop of three edges and thus decreases the edge-count by 3. S splits the current loop in two parts, but if we count the edges in both parts, it increments the total edge count. Each S label starts a recursive call that will fill in the hole bounded by the right loop and will terminate with the corresponding E label. Thus **S and E** labels work as pairs of **parentheses**. Combining all these observation, we can compute the offset by identifying the sub-string of the *clers* string between an S and the corresponding E, and by summing the edge-count changes for each label in that sub-string. To avoid the multiple traversals of the *clers* string, all offsets may be pre-computed by reading the *clers* string once and using a stack for locating the S of each E.

The elegant, "**Spirale Reversi**" approach [IsSn01] for decompressing *clers* strings that have been created by the Edgebreaker compression avoids this preprocessing by reading the *clers* string backwards and building the triangle mesh in reverse order (Fig. 4). It assigns a reference number to a vertex, not at its creation, but only when a C triangle incident upon it is created. The order in which vertices are assigned reference numbers by the Spirale Reversi decompression is reversed from the order in which they are first encountered by the Edgebreaker compression. Note that the vertices of new triangles are initially unlabeled, and remain so, until the corresponding C triangles are created.

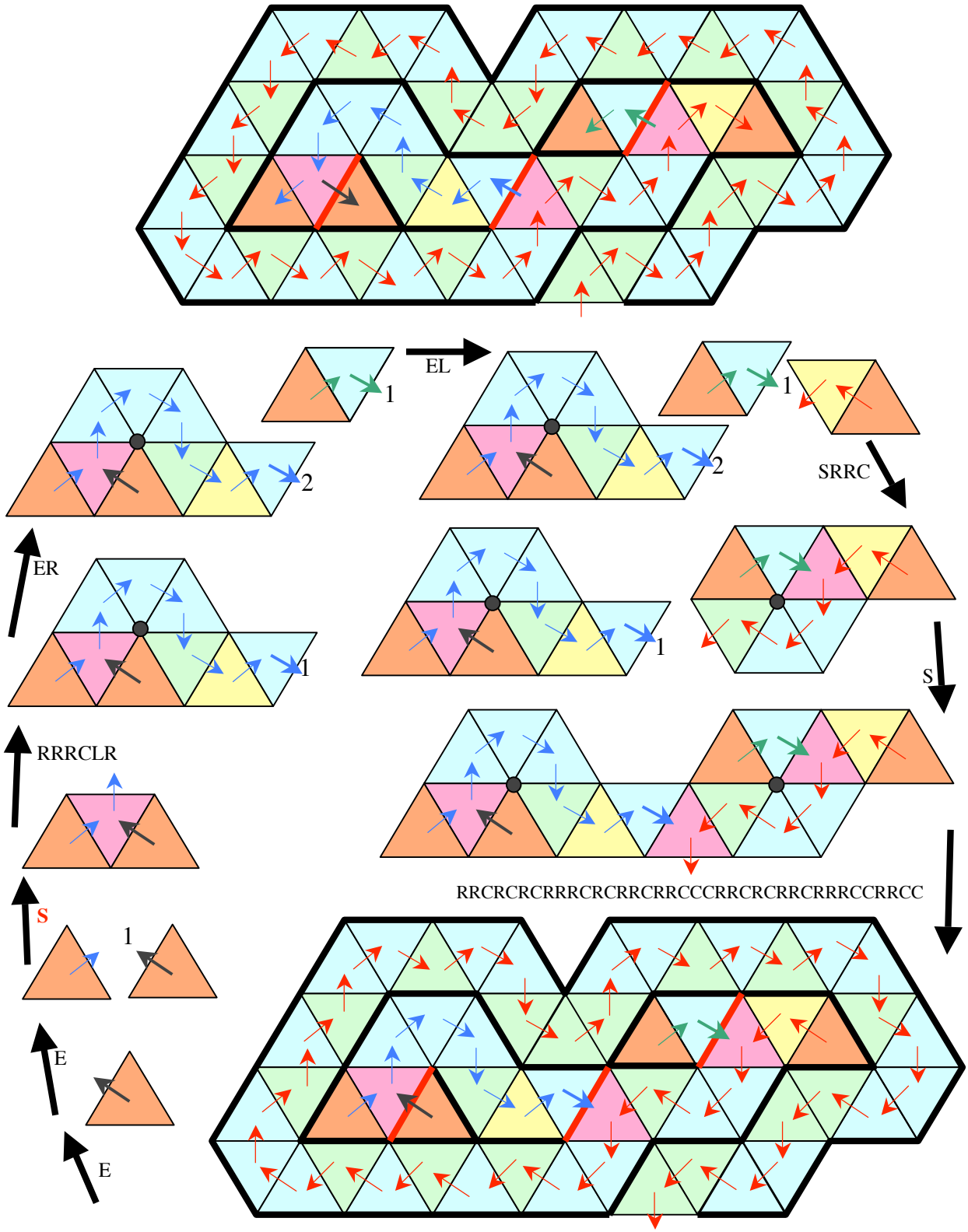


**FIGURE 4:** *The connectivity of the remaining portion of the mesh shown on the top is encoded by Edgebreaker in the clers string: CCRRCCRRRCRRCRCRRCCCRRCRRRCRCRRRCRCRCRRSCRRSLERERLCRRRSEE. The order in which the triangles are visited is shown by the arrows. The Spirale Reversi decompression receives the string reversed. Processing the first symbol (E) of the reversed string generates the first triangle (bottom-left). The arrow leaving the previously reconstructed portion of the mesh indicates the gate where a new triangle will be attached. Then next symbol (another E) puts the gate on a stack (1) and creates a new disconnected triangle with a new gate. Moving clockwise, the next symbol (S) makes a new triangle which joins the gate with the top of the stack. Reading the symbols RRC creates aright-turning fan that encloses a vertex (large dot), which will receive the reference number 1. Then the LR symbols are red. The next symbol (E) puts the gate on the stack (1) and creates an isolated triangle to which another one is attached as we read the next R symbol (top left). This creation, growing, and merging process continues, as shown by the clockwise sequence.*

A third approach, "**Wrap&Zip**" [RoSz99], also avoids the preprocessing of [Ross99] and directly builds a Corner Table as it reads the *clers* string. It does not require maintaining a linked list of border vertices or edges. For each symbol, as a new triangle is attached to the gate, Wrap&Zip fills-in the known entries to the V and O tables. Specifically, it fills in c.o for the tip corner, c, of the new triangle and for its opposite, c.o. It assigns vertex reference numbers to the tips of C triangles as they are created, by simply incrementing a vertex counter. It defers assigning the reference numbers to other vertices until a Zip process matches them with vertices that already have a reference number. Thus, it produces a **web**, as defined earlier. The **border edges** of the web **must be matched** into pairs. The correct matching could be specified by encoding the structure of the cut [Tura84] [TaRo98]. However, as discovered in [RoSz99], the information may be trivially extracted from the *clers* string by **orienting the border edges** of the web as shown in Fig. 5. Note that these border orientations are consistent with an **upward** orientation of the cut-edges toward the root of the **VST.** The pseudo-code for the Wrap&Zip decompression algorithm is shown in the frame below. It was extended to meshes with handles [Lo&02].

```
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

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
```

The zipping part matches pairs of adjacent border edges that are oriented away from their shared vertex. Only the creation of L and E triangles opens new zipping opportunities. Zipping the borders of an E triangle may start a chain of zipping operations (Fig. 6). The cost of the zipping process is linear, since there are as many zipping operations as edges in the VST and the number of failed zipping tests equals the number of E or L triangles.

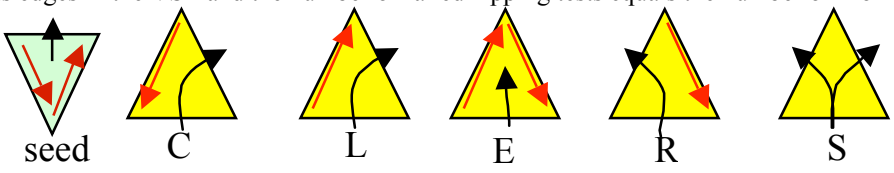


**FIGURE 5:** *The borders of the web are oriented clockwise, except for the seed and the C triangles.*

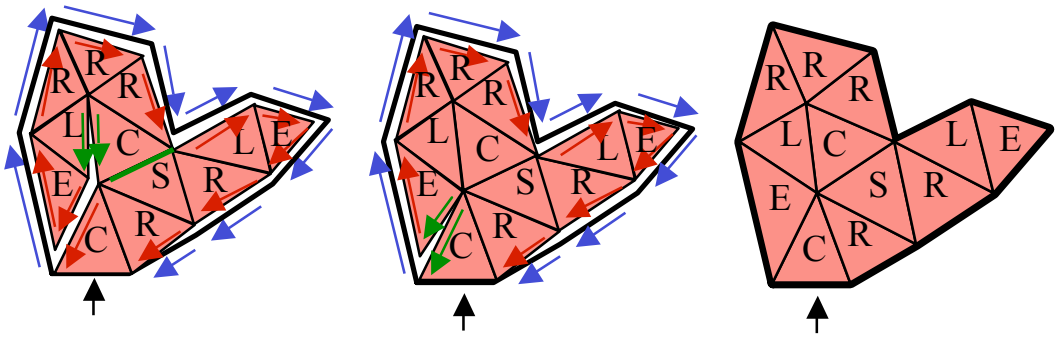


**FIGURE 6:** *We assume that the part of the mesh not shown here has already been decoded into a web with properly oriented borders (exterior arrows). Building the TST(shown by the labeled triangles) for the sub-string CRSRLECRRRLE produces a web whose free borders are oriented clockwise for all non-C triangles and counterclockwise for C triangles (left). Each time Wrap&Zip finds a pair of edges oriented away from their common vertex, it matches them. The result of the first zip operation (center) enables another zip. Repeating the process zips all the borders and restores the desired connectivity (right).*

## Other approaches

The **cut-border machine** [GuSt98] has strong similarities with Edgebreaker. Because it requires the explicit encoding of the offset of S triangles and because it was designed to support manifold meshes with boundaries, cut-border is slightly less effective than Edgebreaker. Reported connectivity compression results range from 1.7$t$ to 2.5$t$ bits. A context-based arithmetic coder further improves them to 0.95$t$ bits [Gumh99]. In [Gumh00] Gumhold proposes a custom variable length scheme that guarantees less than **0.94$t$ bits** for encoding the **offsets**, thus proving that the cut-border machine has linear complexity.

Turan [Tura84] noted that the connectivity of a planar triangle graph can be recovered from the structure of its VST and TST, which he proposed to encode using a total of roughly **12$v$ bits**. Rossignac [Ross99c] has reduced this total cost to **6$v$ bits** by combining two observations: (1) The binary **TST** may be encoded with **2$t$ bits**, using one bit per triangle to indicate whether it has a left child and another one to indicate whether it has a right child. (2) The corresponding (dual) VST may be encoded with 1$t$ bits, one bit per vertex indicating whether the node is a leaf and the other bit per vertex indicating whether it is the last child of its parent. (Remember that $2v=t+4$.) This scheme does not impose any restriction on the TST. Note that for less than the 2$t$ bits budget needed for encoding the TST alone, Edgebreaker [Ross99] encodes the *clers* string, which not only describes how to reconstruct the TST, but also how to orient the borders of the resulting web, so as to define the VST, and hence the complete incidence. This surprising efficiency seems linked to the restriction of using a **spiraling** TST.

Taubin and Rossignac have noticed that a spiraling VST, formed by linking concentric loops into a tree, has relatively **few branches**. Furthermore, the corresponding dual TST, which happens to be identical to the TST produced by Edgebreaker, has also in general few branches (Fig 7). They have exploited this regularity by **Run Length Encoding** (RLE) the **TST** and the **VST**. Each run is formed by consecutive nodes that have a single child. The resulting **Topological Surgery** 3D compression technique [TaRo96, TaRo98] encodes the length of each run, the structure of the trees of runs, and a marching pattern, which encodes each triangle run as a generalized **triangle strip** [ESV96] using one bit per triangle to indicate whether the next triangle of the run is attached to the right or to the left edge of the previous one. An IBM implementation of the Topological Surgery compression has been developed for the **VRML** standard [THLR98] for the transmission of 3D models across the Internet, thus providing a compressed binary alternative to the original VRML ASCII format [VRML97], resulting in a 50-to-1 compression ratio. Subsequently, the Topological Surgery approach has been selected as the core of Three Dimensional Mesh Coding (3DMC) algorithm in **MPEG-4** [MPEG01], which is the ISO/IEC multimedia standard developed by the Moving Picture Experts Group for digital television, interactive graphics, and interactive multimedia applications.

Instead of linking the concentric rings of triangles into a single TST, the **layered structure** color-coded in Fig. 7 (left) may be preserved [BPZ99]. The incidence is represented by the total number of vertex layers, and by the triangulation of each layer. When the layer is simple, its triangulation may be encoded as a **triangle strip**, using one marching bit per triangle, as was originally done in the Topological Surgery approach. However, in practice, a significant number of overhead bits are needed to encode the connectivity of more complex layers. The topological surgery approach resulted from an attempt to reduce this additional cost by chaining the consecutive layers into a single TST (see Fig. 7).

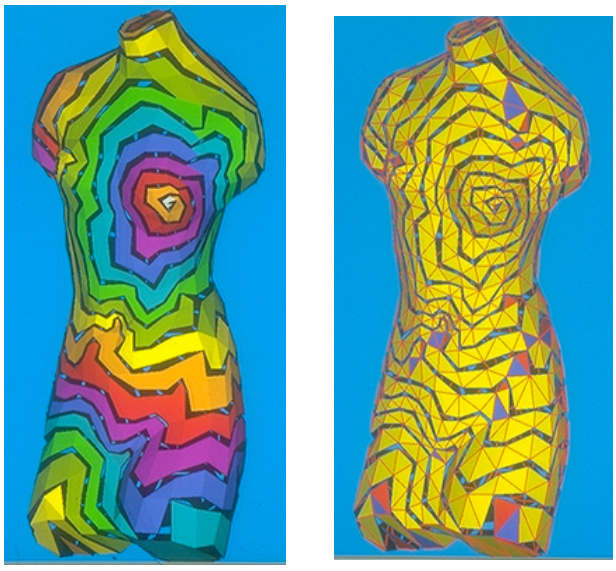

**FIGURE 7:**

*The Topological Surgery approach merges concentric circles of triangles into a single TST (left). That TST and its dual VST have relatively few runs (right).*

Focusing on **hardware** decompression, Deering [Deer95] encodes generalized triangle strips using a buffer of 16 vertices. One bit identifies whether the next triangle is attached to the left or the right border edge of the previous triangle. Another bit indicates whether the tip of the new triangle is encoded in the stream or is still in the buffer and can hence be identified with only 4 bits. Additional bits are used to manage the buffer and to indicate when a new triangle strips must be started. This compressed format is supported by the **Java 3D**'s Compressed Object

node [Java99]. Chow [Chow97] has provided an algorithm for compressing a mesh into Deering's format by extending the border of the previously visited part of the mesh by a fan of not-yet-visited triangles around a border vertex. When the tip of the new triangle is a previously decoded vertex no longer in the cache, its coordinates, or an absolute or relative reference to them, must be included in the vertex stream, significantly increasing the overall transmission cost. Therefore, the optimal encoding traverses a TST that is different from the spiraling TST of Edgebreaker, in an attempt to reduce cache misses.

Given that there are 3 corners per triangle and that $t=2v-4$, there are roughly six times as many corners as vertices. Thus, the **average valence,** i.e., the number of triangles incident upon a vertex, **is 6**. In most models, the valence distribution is highly concentrated around 6. For example, in a subdivision mesh, all vertices, that do not correspond to vertices of the original mesh have valence 6. To exploit this statistics, Touma and Gotsman [ToGo98] have developed a valance-based encoding of the connectivity, which visits the triangles in the same order as Edgebreaker. As in Edgebreaker, they encode the distinction between the C and the S triangles. However, instead of encoding the symbols for L, R, and E, they encode the valence of each vertex and the offset for each S triangle. When the number of incident triangles around a vertex is one less than its valence, the missing L, R, or E triangle may be completed automatically. For this scheme to work, the offset must not only encode the number of vertices separating the gate from the tip of the new triangle along the border (Fig. 2), but also the number of triangles incident on the tip of the S triangle that are part of the right hole. To better appreciate the power of this approach, consider the statistics of a typical case. Only one bit is needed to distinguish a C from an S. Given that 50% of the triangles are of type C and about 5% of the triangles are of type S, the amortized entropy cost of that bit is around **0.22*t*** bits. Therefore, about 80% of the encoding cost lies in the valence, which has a low entropy for regular and finely tessellated meshes and in the encoding of the offsets. For example, when 80% of the vertices have valence 6, a bit used to distinguish them from the other vertices has entropy 0.72 and hence he sequence of these bits may be encoded using close to 0.36*t* bits. The amortized cost of encoding the valence of the other 20% vertices with 2 bits each is 0.40*t* bits. Thus, the **valence** of all vertices in a reasonably regular meshes may be encoded with **0.76*t*** bits. If 5% of the triangles are of type S and each offset is encoded with an average of 5 bits, the amortized cost of the **offsets** reaches **0.25*t* bits**. Note that the offsets add about 25% to the cost of encoding the C/S bits and the valence, yielding a **total of 1.23*t* bits**. This cost drops down significantly for meshes with a much higher proportion of valence-6 vertices.

Although attempts to combine the Edgebreaker solution that avoids sending the offsets and the valence-based encoding of the connectivity have failed, Alliez and Desbrun [AlDe01b] managed to significantly **reduce** the total cost of encoding the offsets, by reducing the **number of S** triangles. They use a heuristic that **selects** as **gate** a border edge incident upon a border vertex with the **maximal number of incident triangles**. To further compress the offsets, they sort the border vertices in the active loop according to their Euclidean distances from the gate and encode the offset values using an arithmetic range encoder [Schi98]. They also show that if one could eliminate the S triangles, the valence-based approach would **guarantee** to compress the mesh with less than **1.62*t* bits**, which happens to be **Tutte's lower bound** [Tutt62].

An improved Edgebreaker compression approach was proposed [SKR00, SKR00b] for sufficiently **large and regular meshes**. It is based on a specially designed context-based coding of the *clers* string and uses the Spirale Reversi decompression. For a sufficiently large ratio of degree six vertices and a sufficiently large *t*, this approach is **proven** to **guarantee** a worst-case storage of **0.81*t* bits**.

# RETILING

Triangle mesh **simplification** techniques, surveyed in [HeGa97, PuSc97] and more recently in [Lu&02], reduce the sampling of the mesh, while decreasing its accuracy. Recently proposed approaches [PoHo97, GaHe97, Lueb98, Lind00] combine ordered edge-collapse operations [Ho&93, Hopp96] with proximity-based or grid-based vertex clustering [RoBo93, LoTa97]. Both merge vertices and eliminate degenerate triangles. Simplification techniques have been developed to accelerate hardware assisted 3D rasterization, and as such, they attempt to find the best compromise between reducing the triangle count and reducing the error. The **error** resulting from such a simplification process may be estimated using the maximum [RoRo96] or the sum [Garl98, GaHe98, Garl99] of the squared distances between the new location of displaced vertices and the planes containing their incident triangles in the original model. It may also be evaluated [CRS98] by sampling the original mesh and computing the distance between the samples and the simplified mesh. One may combine these simplification techniques with the 3D compression approaches described above to achieve a flexible lossy compression. To support the transmission of multi-resolution scenes, several levels-of-detail of each model in the scene should be generated through simplification and compressed. The level at which a model is downloaded may depend on the scale at which a model is projected on the screen and on the total bit budget allocated to the transmission. After an initial transmission of such an approximation of the scene, compressed higher resolution versions of selected models may be downloaded, either to improve the overall accuracy of the image or to adapt to the motions of the viewpoint.

When the complexity ratio between one level-of-detail and the next is large, there is little or no savings in trying to reuse the lower level of detail to reduce the transmission cost of the next level. When finer granularity refinements are desired, parameters to undo the sequence of simplifying edge-collapses in reverse order may be

transmitted one by one [Hopp96], or grouped into 6 to 12 batched and compressed [Hopp98, PaRo00, PaRo00b]. A transmission cost of 3.5*t* bits for connectivity and 7.5*t* bits for geometry was achieved by using one bit per vertex to mark which vertices must be split in each batch [PaRo00, PaRo00b]. Marking the edges, instead of vertices [Vale02, VaPr02, Va&03] allows to recover for free the connectivity of valence-6 portions of the mesh and in general reduces the cost of the progressive transmission of connectivity to between **1.0*t*** bits and **2.5*t*** bits. A series of simplification passes [AlDe01]—which do not take geometry into account, but each divide by 3 the number of vertices through the systematic removal of vertices of valence less than 7, a re-triangulation of the resulting holes, and a subsequent removal of vertices of valence 3—permit to encode connectivity upgrades with about **1.9*t*** bits.

The above simplification or progressive coding techniques encode a simplified version of the original connectivity graph and optionally the data necessary to fully restore it to its original form. When there is no need to preserve that connectivity, one may achieve better compression by producing a more regular sampling of the original mesh at the desired accuracy, so as to reduce the cost of connectivity compression, improve the accuracy of vertex prediction, and reduce the cost of encoding the residues. We review several such retiling techniques.

An early retiling technique [Turk92] first places **samples** on the surface and distributes them evenly using **repulsive forces** derived from estimates of the geodesic distances [PoSc99] between samples. Then it inserts these samples as new vertices in the mesh. Finally, the **old vertices** are **removed** through edge-collapse operations that preserve topology.

The MAPS algorithm [Le&98] was used [KSS00] to compute a **crude simplified** model, which can be compressed using any of the single-rate compression schemes discussed above. Once received and restored by the decompression module, the crude model is used as the coarse mesh of a subdivision process. Each **subdivision stage** splits each edge into two and each triangle into four, by the insertion of one new vertex per edge, in accordance to the **Loop subdivision** [Loop94] rules, which split each edge (c.n.v,c.p.v) by inserting point (c.v.g+c.o.v.g+3c.n.v.g+3c.o.v.g)/8 and then displaces the old vertices towards the average of their old neighbors. After each subdivision stage, the client downloads a displacement field of **corrective vectors** and uses them to adjust the vertices, so as to bring the current level subdivision surface closer to the desired surface. The distribution of the coefficients of the corrective vectors is concentrated around zero and their magnitude diminishes as subdivision progresses. They are encoded using a wavelet transform and compressed using a modified version of the SPIHT algorithm [SaPe96] originally developed for image compression.

Instead of encoding corrective 3D vectors, the **Normal Mesh** approach [GVSS00] restrict each offset vector to be parallel to the surface normal estimated at the vertex. Only one corrective displacement value needs to be encoded per vertex, instead of three coordinates. A **Butterfly subdivision** [DLG90] is used. It preserves the old vertices, and for each pair of opposite corners c and c.o splits edge (c.n.v,c.p.v) by inserting point (8c.n.v.g+8c.p.v.g+2c.v.g+2c.o.v.g+c.l.v.g+c.r.v.g+c.o.l.v.g+c.o.r.v.g)/16. The corrective displacement of new vertices are compressed using the un-lifted version of the Butterfly **wavelet transform** [DLG90, ZSS96]. Further subdivision stages generate a smoother mesh that interpolates these displaced vertices. The challenge of this approach lies in the computation of a suitable crude simplified model and in handling situations where no suitable displacement for a new vertex exists along the estimated surface normal. The connectivity of the crude mesh and the constraint imposed by the regular subdivision process limit the way in which the retiling can adapt to the local shape characteristics, and thus may result in suboptimal compression ratios. For example, regular meshes may lead to **sub-optimal triangulations** for surfaces with high curvature regions and saddle points, where vertices of valence different than 6 would be more appropriate.

In the PRM approach [SRK02], the surface may be algorithmically decomposed into 6 **reliefs,** each one comprising triangles whose normals are closest to one of the six principal directions (Fig. 8 left). Each relief is re-sampled along a regular grid of parallel rays (Fig. 8 right). Triangles are formed between samples on adjacent rays and also, to ensure the proper connectivity, at the junction of adjacent reliefs. When the sampling rate (i.e. the density of the rays) is chosen so that the resulting **Piecewise Regular Mesh** (**PRM**) has roughly the same number of vertices as the original mesh, the PRM approximates the original mesh with the mean square error of less than 0.02% of the diameter of the bounding box. Because of the regularity of the sampling in each relief, the PRM may be compressed down to a total about **2*t* bits**, which accounts for both **connectivity and geometry**. PRM uses Edgebreaker compression [Ross99] and the Spirale Reversi decompression [IsSn01] to encode the global relief connectivity and the triangles which do not belong to the regular regions. Edgebreaker produces the CLERS string, which is then turned into a binary string using the **context-based range coder**, which reduces the uncertainty about the next symbol for a highly regular mesh. The **geometry** of the reliefs is compressed using an iterated two-dimensional variant of the **differential coding**. The regular retiling causes the entropy of the parallelogram rule residues to decrease by about 40% when compared to the entropy for the original models, because, on reliefs, two out of three coordinates of the residual vectors become zero. Since this approach does not require global parameterization, it may be used for models with **complex topologies**. It is faster than the combination of the MAPS algorithm [Le&98] and the wavelet mesh compression algorithm of [GVSS00, KSS00], while offering comparable compression rates.

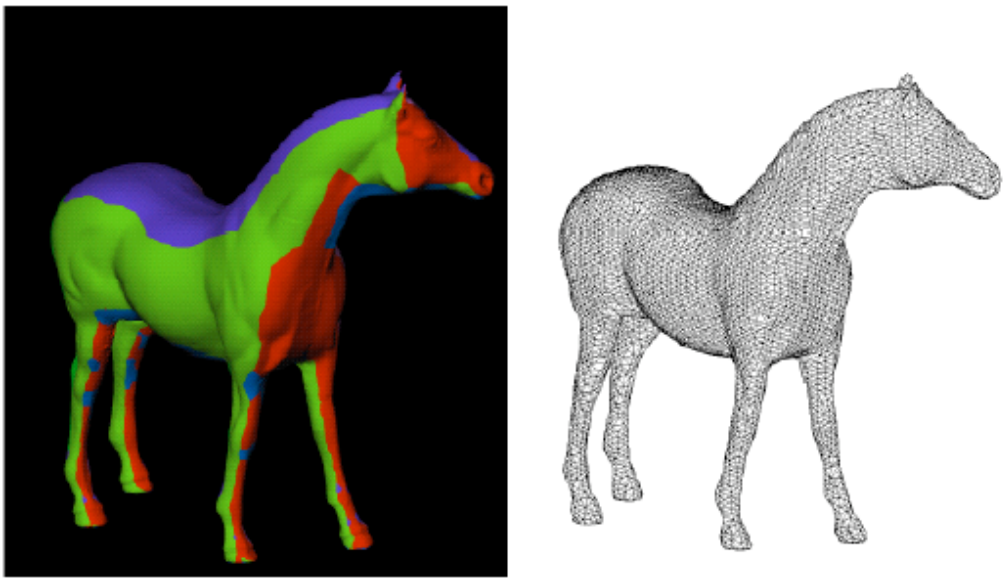

**FIGURE 8:** The reliefs produced by the Piecewise Regular Mesh (PRM) approach are shown (left) and resampled (right) into a nearly regular triangle mesh.

By tracing geodesics, SwingWrapper [Atte01] partitions the surface of an original mesh M into simply connected regions, called triangloids. From these, it generates a new mesh M'. Each triangle of M' is a linear approximation of a triangloid of M (Fig. 9). By construction, the connectivity of M' is fairly regular (96% of the triangles are of type C or R and 82% of the vertices have valence 6) and can be compressed to less than a bit per triangloid using Edgebreaker. The locations of the vertices of M' are encoded with about 6 bits per vertex of M', thanks to a new prediction technique, which uses a single correction parameter per vertex. Instead of using displacements along the surface normal or heights on a grid of parallel rays, SwingWrapper requires that the left and right edges of all C triangles have a prescribed length L, so that the correction to the predicted location of their tip may be encoded using the angle of the hinge edge. For a variety of popular models, it is easy to create an M' with 10 times fewer triangles than M. the appropriate choice of L yields a total file size of 0.4*t* bits and a mean square error with respect to the original of about 0.01% of the bounding box diagonal.

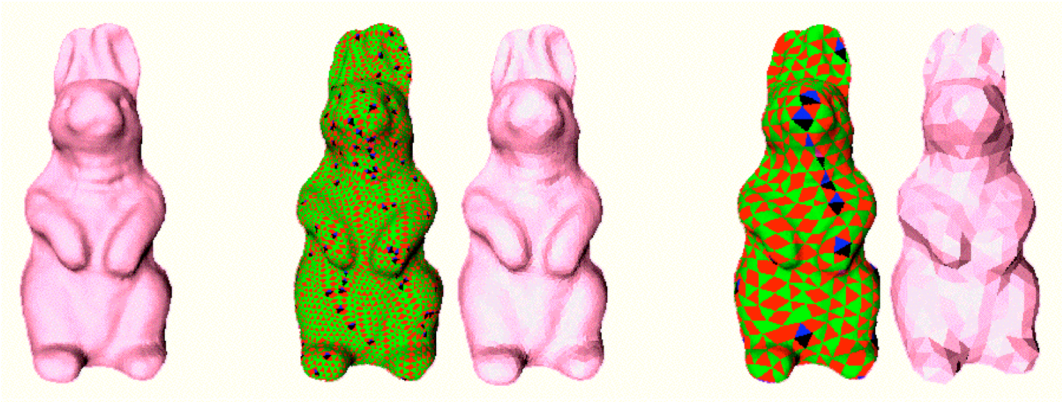

**FIGURE 9***: The original model (first from the left, courtesy of Cyberware) contains t=134,074 triangles. A dense partitioning of its surface into triangloids (second), was produced by SwingWrapper. The corresponding retiled mesh (third) was generated by flattening the triangloids. Its $L^2$ error is about 0.007% of the bounding box diagonal and its 13642 triangles were encoded with a total of 3.5 bits per triangloid, for both the connectivity and geometry, using Edgebreaker's connectivity compression combined with a novel geometry predictor, yielding a compressed file 0.36t bits. A coarser partitioning (fourth) decomposes the original surface into 1505 triangloids. The distortion of the corresponding retiled mesh (last) is about 0.15%, the total file size is 0.06t bits.*

# CONCLUSION

The connectivity of triangle meshes homeomorphic to a sphere may always be encoded with less than 1.8*t* bits and usually requires less than 1.0*t* bits. Their quantized geometry may be compressed to about 7.0*t* bits. Compression and decompression algorithms are extremely simple and efficient. Progressive transmission doubles the total cost of connectivity. When the original connectivity needs not be preserved, retiling the surface of the mesh to enhance regularity often reduces the storage cost to 0.5*t* bits with a mean square error of less than 1/10000 of the size of the model. These statistics remain valid for meshes with relatively few holes and handles.

# REFERENCES

[AlDe01] P. Alliez and M. Desbrun, "Progressive encoding for lossless transmission of 3D meshes," in ACM SIGGRAPH Conference Proceedings, 2001.

[AlDe01b] P. Alliez and M. Desbrun, "Valence-driven connectivity encoding for 3D meshes," EUROGRAPHICS, vol. 20, no. 3, 2001.

[Atte01] M. Attene, B. Falcidieno, M. Spagnuolo, J. Rossignac, "SwingWrapper: Retiling Triangle Meshes for Better EdgeBreaker Compression", Genova CNR- IMA Tech. Rep. No. 14/2001. To appear in the ACM Transactions on Graphics. Volume 22, No. 4, October 2003.

[BPZ99] C. L. Bajaj, V. Pascucci, and G. Zhuang, "Single resolution compression of arbitrary triangular meshes with properties," Computational Geometry: Theory and Applications, vol. 14, pp. 167–186, 1999.

[Chow97] M. Chow, "Optimized geometry compression for real-time rendering," in Proceedings of the Conference on Visualization '97, pp. 347–354, 1997.

[CoRo02] V. Coors and J. Rossignac, "Guess Connectivity: Delphi Encoding in Edgebreaker", GVU Technical Report, 2002.

[CRS98] P. Cignoni, C. Rocchini and R. Scopigno, "Metro: measuring error on simplified surfaces", Proc. Eurographics '98, vol. 17(2), pp 167-174, June 1998.

[Deer95] M. Deering, "Geometry compression," in Proceedings of the 22nd Annual ACM Conference on Computer Graphics, pp. 13–20, 1995.

[DLG90] N. Dyn, D. Levin, and J. A. Gregory, "A butterfly subdivision scheme for surface interpolation with tension control," ACM Transactions on Graphics, vol. 9, no. 2, pp. 160–169, 1990.

[ESV96] F. Evans, S. S. Skiena, and A. Varshney, "Optimizing triangle strips for fast rendering," in IEEE Visualization '96, 1996, pp. 319–326.

[GaHe97] M. Garland and P. Heckbert, "Surface simplification using quadric error metrics", Proc. ACM SIGGRAPH'97. pp. 209-216. 1997.

[GaHe98] M. Garland and P. Heckbert, "Simplifying Surfaces with Color and Texture using Quadratic Error Metric". Proceedings of IEEE Visualization, pp. 287-295, 1998.

[Garl98] M. Garland. "Quadric-based Polygonal Surface Simplification". PhD Thesis, Carnegie Mellon University, 1998.

[Garl99] M. Garland, "QSlim 2.0" [Computer Software]. University of Illinois at Urbana-Champaign, UIUC Computer Graphics Lab, 1999. xhttp://graphics.cs.uiuc.edu/~garland/software/qslim.html.

[GBTS99] A. Gueziec, F. Bossen, G. Taubin, and C. Silva, "Efficient Compression of Non-manifold Polygonal Meshes". In IEEE Visualization, pp. 73–80, 1999.]

[Gumh00] S. Gumhold, "Towards optimal coding and ongoing research", in 3D Geometry Compression, Course Notes, SIGGRAPH 2000.

[Gumh99] S. Gumhold, "Improved cut-border machine for triangle mesh compression," in Erlangen Workshop '99 on Vision, Modeling and Visualization. IEEE Signal Processing Society, Nov. 1999.

[GuSt98] S. Gumhold and W. Straßer, "Real time compression of triangle mesh connectivity," in Proceedings of the 25th Annual Conference on Computer Graphics, 1998, pp. 133–140.

[GVSS00] I. Guskov, K. Vidimce, W. Sweldens, and P. Schroeder, "Normal meshes," in Siggraph'2000 Conference Proceedings, July 2000, pp. 95–102.

[HeGa97] Paul Heckbert and Michael Garland. "Survey of polygonal simplification algorithms". In Multi-resolution Surface Modeling Course. ACM SIGGRAPH Course Notes, 1997.

[Ho&93] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, "Mesh optimization," in Computer Graphics: Siggraph '93 Proceedings, 1993, pp. 19–25.

[Hopp96] H. Hoppe, "Progressive meshes". Computer Graphics, vol. 30, no. Annual Conference Series, pp. 99–108, 1996.

[Hopp98] H. Hoppe, "Efficient implementation of progressive meshes," Computers and Graphics, vol. 22, no. 1, pp. 27–36, 1998.

[IsSn00] M. Isenburg and J. Snoeylink, "Face fixer: Compressing polygon meshes with properties," in Siggraph 2000, Computer Graphics Proceedings, 2000, pp. 263–270.

[IsSn01] M. Isenburg and J. Snoeyink, "Spirale Reversi: Reverse decoding of the Edgebreaker encoding", Computational Geometry, vol. 20, no. 1, pp. 39-52, October 2001.

[Java99] Sun Microsystems. "Java3d API Specification". http://java.sun.com/products/java-media/3D, June 1999.

[KaTa96] A.D. Kalvin and R.H. Taylor, "Superfaces: Polygonal mesh simplification with bounded error". IEEE Computer Graphics and Applications, 16(3), pp. 64-67, 1996.

[KiRo99] 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.

[KRS99] D. King, J. Rossignac, and A. Szymczak, "Connectivity compression for irregular quadrilateral meshes," Technical Report TR–99–36, GVU, Georgia Tech, 1999.

[KSS00] A. Khodakovsky, P. Schroeder, and W. Sweldens, “Progressive geometry compression,” in SIGGRAPH 2000, Computer Graphics Proceedings, 2000, pp. 271–278.

[Le&98] A. W. F. Lee, W. Sweldens, P. Schroeder, L. Cowsar, and D. Dobkin, “MAPS: Multiresolution adaptive parametrization of surfaces,” in SIGGRAPH’98 Conference Proceedings, 1998, pp. 95–104.

[Lind00] P. Lindstrom, “Out-of-core simplification of Large Polygonal Models”. Proc. ACM SIGGRAPH, pp. 259-262, 2000.

[Lo&02] H. Lopes, J. Rossignac, A. Safanova, A. Szymczak and G. Tavares. “A Simple Compression Algorithm for Surfaces with Handles”, ACM Symposium on Solid Modeling, Saarbrucken. June 2002.

[Loop94] C. Loop, “Smooth Spline Surfaces over Irregular Meshes” , Computer Graphics, Vol. 28, Number Annual Conference Series, pp. 303-310, July 1994.

[LoTa97] K-L. Low and T.S. Tan, “Model Simplification using vertex clustering”, Proc. Symp. Interactive 3D Graphics, ACM Press, NY, pp. 75-82, 1997.

[Lu&02] D. Luebke, M Reddy, J. Cohen, A. Varshney, B. Watson, R. Hubner, “Levels of Detail for 3D Graphics”, Morgan Kaufnamm, 2002.

[Lueb98] D. P. Luebke, “View-dependent simplification of arbitrary polygonal environments”, Doctoral Dissertation, University of North Carolina at Chapel Hill, 1998. http://www.cs.virginia.edu/~luebke/simplification.html.

[MPEG01] ISO/IEC 14496-2, “Coding of Audio-Visual Objects: Visual”, July 2001.

[PaRo00] R. Pajarola and J. Rossignac, “Compressed progressive meshes,” IEEE Transactions on Visualization and Computer Graphics, vol. 6, no. 1, pp. 79–93, 2000.

[PaRo00b] R. Pajarola and J. Rossignac, “Squeeze: Fast and progressive decompression of triangle meshes,” in Proceedings of Computer Graphics International Conference, 2000, pp. 173–182.

[PoHo97] J. Popovic and H. Hoppe, “Progressive simplicial complexes,” Computer Graphics, vol. 31, pp. 217–224, 1997.

[PoSc99] K. Polthier, M. Schmies, “Geodesic Flow on Polyhedral Surfaces”, Procs. of Eurographics Workshop on Scientific Visualization, Vienna 1999.

[PRS99] R. Pajarola, J. Rossignac, and A. Szymczak, “Implant Sprays: Compression of Progressive Tetrahedral Mesh Connectivity”, IEEE Visualization 1999, San Francisco, October 24-29, 1999.

[PuSc97] E. Puppo and R. Scopigno, “Simplification, LOD and multiresolution: Principles and applications”, Tutorial at the Eurographics’97 conference, Budapest, Hungary, September 1997.

[RoBo93] J. Rossignac and P. Borrel, “Multi-resolution 3D approximations for rendering complex scenes”, Geometric Modeling in Computer Graphics, Springer Verlag, Berlin, pp. 445-465, 1993.

[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.

[RoRo96] R. Ronfard and J. Rossignac, “Full range approximation of triangulated polyhedra”, Proc. Eurographics 96, 15(3), pp. 67-76, 1996.

[Ross99] J. Rossignac, “Edgebreaker: Connectivity compression for triangle meshes,” IEEE Transactions on Visualization and Computer Graphics, vol. 5, no. 1, pp. 47–61, 1999.

[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.

[RSS01] J. Rossignac, A. Safonova, and A. Syzmczak, "3D Compression Made Simple: Edgebreaker on a Corner-Table", Invited lecture at the Shape Modeling International Conference, Genoa, Italy, May 2001.

[RSS02] J. Rossignac, A. Safonova, A. Szymczak “Edgebreaker on a Corner Table: A simple technique for representing and compressing triangulated surfaces”, in Hierarchical and Geometrical Methods in Scientific Visualization, Farin, G., Hagen, H. and Hamann, B., eds. Springer-Verlag, Heidelberg, Germany, to appear in 2002.

[Salo00] D. Salomon, “Data Compression: The Complete Reference”. Second Edition, Springer Verlag Berlin Heidelberg. 2000.

[SaPe96] A. Said and W. A. Pearlman, “A new, fast, and effcient image codec based on set partitioning in hierarchical trees,” IEEE Trans. Circuits Syst. Video Technol., vol. 6, no. 3, pp. 243–250, June 1996.

[SaRo02] A. Safonova and J. Rossignac, Source code for an implementation of the Edgebreaker compression and decompression www.gvu.gatech.edu/~jarek/edgebreaker/eb

[Schi98] M. Schindler, “A fast renormalization for arithmetic coding,” in Proceedings of IEEE Data Compression Conference, 1998, p. 572.

[SKR00] A. Szymczak, D. King, and J. Rossignac, “An Edgebreaker-based efficient compression scheme for regular meshes,” in Proceedings of 12th Canadian Conference on Computational Geometry, 2000, pp. 257–264.

[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.]

[SRK02] A. Szymczak, J. Rossignac, and D. King. “Piecewise Regular Meshes: Construction and Compression.” To appear in Graphics Models, Special Issue on Processing of Large Polygonal Meshes, 2002.

[SzRo00] A. Szymczak and J. Rossignac. "Grow&Fold: Compressing the connectivity of tetrahedral meshes", Computer-Aided Design. 32(8/9), 527-538, July/August, 2000.

[SzRo99] A. Szymczak and J. Rossignac. "Grow&Fold: Compression of Tetrahedral Meshes", Proc. ACM Symposium on Solid Modeling, June 1999, pp. 54-64

[TaRo96] G. Taubin and J. Rossignac, “Geometric Compression through Topological Surgery”. IBM Research Report RC-20340. January 1996. (http://www.watson.ibm.com:8080/PS/7990.ps.gz).

[TaRo98] G. Taubin and J. Rossignac, “Geometric compression through topological surgery,” ACM Transactions on Graphics, vol. 17, no. 2, pp. 84–115, 1998.

[THLR98] G. Taubin, W. Horn, F. Lazarus, and J. Rossignac, “Geometry coding and VRML,” Proceeding of the IEEE, vol. 96, no. 6, pp. 1228–1243, June 1998.

[ToGo98] C. Touma and C. Gotsman, “Triangle mesh compression,” in Graphics Interface, 1998, pp. 26–34.

[Tura84] G. Turan, “On the succinct representations of graphs,” Discrete Applied Mathematics, vol. 8, pp. 289–294, 1984.

[Turk92] G. Turk, “Retiling polygonal surfaces”, Proc. ACM Siggraph 92, pp. 55-64, July 1992.

[Tutt62] W. Tutte, “A census of planar triangulations”, Canadian Journal of Mathematics, pp.21-38, 1962.

[Va&03] S. Valette, J. Rossignac, R. Prost. “An efficient subdivision inversion for Wavemesh-based progressive compression of 3d triangle meshes”, Submitted for publication. 2003.

[Vale02] S. Valette “Modeles de maillage deformables 2D et multiresolution surfacique 3D sur une base d’ondelettes”. Doctoral Dissertation. January 2002. INSA Lyon.

[VaPr02] S. Valette and R. Prost, “A Wavelet-based Progressive Compression Scheme for Triangle Meshes: Wavemesh”, submitted for publication, September, 2002.

[VRML97] ISO/IEC 14772-1, The Virtual Reality Modeling Language (VRML), 1997.

[ZSS96] D. Zorin, P. Schroeder, and W. Sweldens, “Interpolating subdivision for meshes with arbitrary topology,” Computer Graphics, vol. 30, pp. 189–192, 1996.