

Edgebreaker compression and Wrap&Zip decoding of the connectivity of triangle meshes

Jarek Rossignac and Andrzej Szymczak
GVU Center, Georgia Institute of Technology

Abstract

The Edgebreaker compression, introduced by Rossignac in [13], encodes any unlabeled triangulated planar graph of t triangles using a CLERS string of $2t$ bits, which represents a sequence of t symbols from the set $\{C, L, E, R, S\}$. Exploiting constraints between consecutive symbols, the CLERS string may in practice be encoded with $1.3t$ bits using simple codes or further compressed to $0.9t$ bits using an entropy code. These results improve over the $2.3t$ bits code of Keeler and Westbrook [8] and over the various 3D triangle mesh compression techniques published recently [5, 7, 11, 20, 21], which exhibit either larger constants or cannot guarantee a linear worst case storage complexity. We compress the mesh using the same spiraling triangle-spanning tree as [13] and generate the same CLERS string. Edgebreaker's decompression uses a look-ahead procedure to identify the third vertex of split-triangles (S type) by counting letter occurrences in the remaining part of the CLERS string. To avoid this cost and the associated non-linear decompression time complexity, we introduce here a new decompression technique, called Wrap&Zip. It converts the string into the corresponding triangle-spanning tree and assigns orientations to each one of its bounding edges. Whenever two consecutive bounding edges point to the same vertex, it glues them together. By labeling the vertices according to the order in which they first appear in the triangle-spanning tree, this graph encoding may be used to encode the connectivity (incidence of labeled graphs) of three-dimensional triangle meshes that are homeomorphic to a sphere. We also provide simple extensions to more general triangle meshes with holes and handles.

Introduction

Planar graphs

We consider first planar triangle graphs of t triangles and v vertices. These are topologically equivalent to the connectivity graph of a triangulated surface that is homeomorphic to a sphere. Note that $t=2v-4$ for such graphs. In this paper, we use the term *simple mesh* to refer to such a graph. We then extend our work to meshes that may be represented as topological manifolds with zero or more handles (i.e., through holes) and bounding loops (i.e., holes in the surface).

Previously reported compression schemes

The connectivity of a simple mesh may be stored as a sequence of t triangle descriptors, each triangle been represented by 3 integer labels. Each labels identifies one amongst the v vertices and requires $\lceil \log_2(v) \rceil$ bits. (From now on, we will assume that the ceiling notation " $\lceil \cdot \rceil$ " is implicit for all log expressions.) Organizing triangles into strips [4], where each new triangle shares an edge with the previous one, reduces in practice the above storage by half. The use of a buffer to cache a small number of labels [2] may further reduce the expected cost.

To encode the structure of a labeled planar graph, Turan [21] builds a vertex-spanning tree which represents the boundary of a topological polygon of $2v-2$ edges. The structure of the vertex-spanning tree is encoded with $4v-4$ bits. There are at most $2v-5$ edges that do not belong to the vertex-spanning tree. These may be encoded using 4 bits each. The overall connectivity cost is thus, $12v-24$ bits.

Inspired by Tutte [22], Itai and Rodeh [7] show that any unlabeled rooted non-separable triangulated planar graphs of v vertices may be represented by $4v$ bits. They also propose a

linear algorithm for constructing a representation of any labeled planar graph using at most $1.5v\log_2(v)+6v+O(\log_2(v))$ bits, while the theoretical minimum is $v\log_2(v)+O(v)$. They use a triangle as the initial outer loop and then shrink that loop by removing one triangle at a time. They always delete the triangle that is incident to the smallest vertex v_1 in the outer loop and is bounded by the outer loop edge that starts at v_1 . They distinguish four cases: (1) The third vertex precedes v_1 in the outer loop; (2) It follows the successor of v_1 ; (3) It is somewhere else in the outer loop; and (4) it is not on the outer loop. Operations (3) and (4) each require $\log_2(v)$ bits to identify a vertex in the not yet processed part of the mesh. Several improvements over Itai and Rodeh's method were reported recently [5, 13, 14, 17, 18].

Cutting through the edges of the vertex-spanning tree produces a triangulated, simply connected surface without internal vertices. It may be completely represented by the triangle-spanning tree, which is a binary tree, whose nodes correspond to the triangles and whose edges correspond to some of the edges of the mesh. A depth-first traversal of such a spanning tree corresponds to a walk on the entire mesh that starts at the root triangle and recursively visits the neighboring triangles that have not been previously visited. The triangle-spanning tree may be encoded using $2t$ bits, but does not contain sufficient information to recover the mesh.

The Topological Surgery method of Taubin and Rossignac [17, 18] compress both a triangle-spanning tree and its dual vertex-spanning tree by encoding the lengths of consecutive single-child nodes. Both trees suffice to encode the connectivity of the simple mesh. For complex and reasonably regular meshes, the expected cost of encoding both trees may amount to about two bits per triangle. However, the overhead

of the run length encoding may result in a significantly higher average cost for irregular or small meshes.

Rossignac [14] has proposed a variation, which uses 2 bits per vertex to encode the vertex-spanning tree (one bit indicates the presence of a child while the other bit indicates the presence of a right sibling) and 2 bits per triangle to encode the triangle-spanning tree (one bit indicates the presence of a right child, while the other bit indicates the presence of a left child). With twice more triangles than vertices, the guaranteed worst case connectivity cost of this representation is $3t$ bits.

Gumhold and Strasser's technique [5] and Rossignac's Edgebreaker scheme [13], although developed independently, are closely related. Both schemes perform the same traversal of the mesh as in [17]. At each step, they remove a triangle and encode the necessary information to reconstruct the triangle by distinguishing several cases that include the four cases of Itai and Rodeh. Edgebreaker uses the letters L, R, S, and C to identify cases 1 through 4 of Itai and Rodeh. Gumhold and Strasser add the case where a boundary edge is reached. Edgebreaker does not need to distinguish this case, since it encodes the bounding loop at the beginning of the vertex array. However, Edgebreaker adds the case E, which corresponds to the situation where the current triangle is not adjacent to any other remaining triangle. Both these approaches avoid the $\log_2(v)$ bits cost associated with case (4) of Itai and Rodeh by encoding the vertices in the order in which they are visited by case (4). However, with each case (3) operation, Gumhold and Strasser must encode the reference to a vertex in the current boundary, which requires $\log_2(v)$ bits and makes their storage costs a non-linear function of v .

Instead, Edgebreaker uses a decompression preprocessing step to compute these vertex-references from the sequence of symbols, and therefore exhibits a linear storage cost, although a non-linear time complexity. For typical meshes, Gumhold and Strasser report compression results between 1.7 and 2.15 bits per triangle using Huffman encoding of the bit stream.

Keeler and Westbrook [8] improve on Turan's results and propose a technique for encoding planar graphs with a guaranteed $4.6v$ bits. They also build a triangle-spanning tree. Each triangle of the tree, except the root, shares an edge with its parent and may have zero, one, or two children and thus two, one, or zero free edges. They append free edges to the leaves of the triangle-spanning tree and label them. Encoding the graph and the labels requires an average of $1 + \log_2(3)/3$ bits per edge. They suggest a coding scheme based on a series of graph transformations.

Touma and Gotsman [20] also encode the vertices along the vertex-spanning tree in the same spiraling order as [5,13,17]. They distinguish only two cases, which correspond to the cases (3) and (4) of Itai and Rodeh and to the Edgebreaker's cases S and C. Other cases are not encoded. Instead, Touma and Gotsman encode the degree of each vertex, i.e., the number of incident edges and use it to automatically identify the other cases. During decompression, they keep track of the number of already decoded triangles that are incident upon each vertex and are thus capable of identifying the R, L, and E triangles

automatically. For highly tessellated regular models, where the degree of the vertices follows almost regular patterns, they report compression results of less than a bit per triangle using Huffman encoding. However, for smaller or less regular meshes, the required storage may easily exceed 2 bits per triangle. As Itai and Rodeh and as Gumhold and Strasser, they require that with each S operation be associated a vertex reference, which requires $\log_2(v)$ bits, prior to Huffman compression, and makes the worst case storage cost a non-linear function of v .

The non-linear worst case behavior of the algorithms described in [5, 6, 20] may be illustrated in Fig. 1.

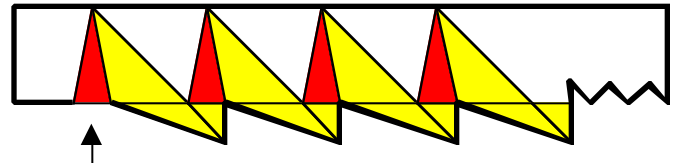


Figure 1: Repeating the above sequence, we can create a model where 20% of the triangles are split triangles (marked in red). If each split triangle is associated with an integer offset value ranging from 0 to the number of vertices in the current loop (thick black line), which may contain a number of vertices proportional to t , the worst case storage cost is $O(t \log_2(t))$. Furthermore, the time complexity of the associated decompression algorithms is $O(t^2)$, if they traverse an unbounded portion of the current loop to identify the tip of each red triangle.

Inspired by [9] and improving on [11, 15], Denny and Sohler have proposed a technique for encoding the incidence of 2D triangulations of sufficiently large size as a permutation of the vertices [3]. They show that there are less than $2^{8.2v + O(\log v)}$ valid triangulations of a planar set of v points, and that for sufficiently large v , each triangulation may be associated with a different permutations of these points (there are approximately $2^{v \log(v)}$ such permutations). They transmit the suitably ordered vertices in batches. The decompression sorts them lexicographically, computes a permutation number by comparing the order in which the vertices were received with their lexicographic order, then sweeps over the previously recovered triangulation from left to right and refines it by inserting the new vertices. At each vertex of the current batch, it identifies the enclosing triangle [10] and the vertex is inserted according to the incidence relation derived from the bit string that encodes the permutation number. Unfortunately, the unstructured order in which the vertices are received and the absence of the incidence graph during their decompression makes it difficult to combine this approach with the predictive techniques for vertex encoding, which are fundamental in mesh compression and rely on a priori knowledge of incidence.

Recently, Chuang et al. and He et al. [1,6] have discovered an encoding of $4v-9$ bits for planar triangle graphs that has linear time compression and decompression complexity. It uses a parentheses-based encoding of a vertex-spanning tree constructed using a canonical ordering and labels other edges with one bit per edge.

The Edgebreaker compression

The Edgebreaker compression algorithm was introduced in [13]. We include it here for completeness and attempt to explain it in a simpler and more concise manner, which may help readers develop more effective implementations.

Spiraling triangle-spanning tree

The Edgebreaker compression algorithm visits the triangles of the mesh in the order in which they appear in a spiraling triangle-spanning tree. Such a tree may be built efficiently by the recursive procedure described below, but may also be built by a trivial recursive procedure, which visits adjacent triangles and marks them. Such a traversal tends to construct an imperfect spiral of triangles.

During the recursive traversal, we leave a triangle P to enter an adjacent triangle X that has not been previously visited. P is the parent of X in the triangle-spanning tree. The other two triangles adjacent to X may be consistently identified as X.left and X.right, using a convention and a consistent orientation of the triangles throughout the mesh. If X.right has not been previously visited, it is appended as the right child of X and is visited by the recursive procedure in a depth-first order. Then, if X.left has not been visited, it is appended as the left child of X and is also visited by the recursion. We mark all the vertices of visited triangles.

The procedure starts with any triangle X of the mesh and identifies one of its edges as the starting gate, which suffices to define X.right and X.left.

Let v be the only vertex of X that is not the vertex of its parent in the tree. The “visited or not-visited” status of v , of X.left, and of X.right unambiguously identifies one out of the five possible situations depicted in Fig. 2 and associated with the letters: C, L, E, R, and S. Compression simply encodes the corresponding letters—or more precisely their binary op-codes—in the order in which the corresponding triangles are visited. The resulting CLERS string of t symbols represents a compact encoding of the connectivity of the mesh.

The selection of the appropriate case may be performed by the following sequence of tests:

```

if not visited( $v$ ) then C
  else if visited(X.left)
    then if visited(X.right) then E else L
    else if visited(X.right) then R else S
    
```

If the vertices are labeled in the order in which they are first visited by this traversal and then encoded in the order of increasing labels, this approach may be used to compress labeled graphs and 3D triangle meshes homeomorphic to a sphere [13].

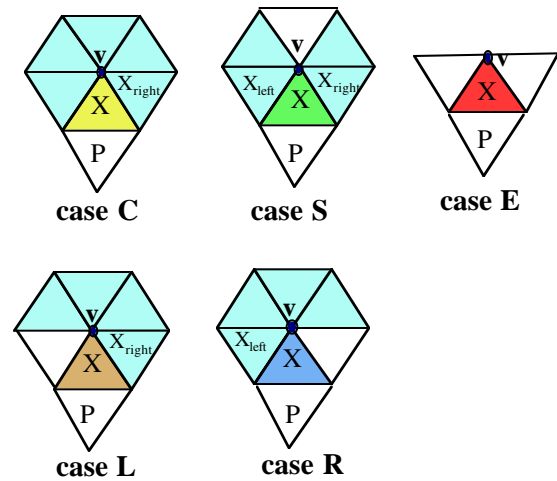


Figure 2: Previously visited triangles are not shaded.

- Case C: v has not been visited.
- Case S: Only v has been visited.
- Case E: X.left and X.right have been visited.
- Case L: Only X.right was not visited.
- Case R: Only X.left was not visited.

Efficient encoding of the op-codes

Guaranteed 2t bit code

Except for the first two vertices, there is a one-to-one association between the vertices of the mesh and the triangles processed by C operations. Therefore, the number of Cs is $v-2$, which equals $t/2$, given that $v=(t+4)/2$. The total number of non-C operations, $t-v+2$, also equals $t/2$. Hence, if we use a 1-bit code for C and 3-bit codes for the other four operations, the total cost for storing the string with the above scheme would be exactly $2t$.

Because the first two operations are Cs, they can be omitted from the string, yielding a total storage cost of $2t-2$ bits for any triangular planar graph.

Expected 1.7t bit code not exceeding 2t bits

Because CL and CE combinations are impossible, we can use a shorter code for S and R symbols that follow a C, thus we have two modes: "after a C" and "not after a C". In the "after a C" mode, there are only 3 possible symbols: C (coded as 0), R (coded as 11) and S (coded 10). In the "not after a C" mode use the following codes for the five possible symbols: 0 for C, 110 for L, 101 for R, 100 for E, and 111 for S. Experimental results with this code (which show a ratio of 36% R operations, almost half of which follow a C) consistently result in a storage cost of 1.7t bits. The code will never exceed 2t bits.

Expected 1.3t to 1.6t bit code

By exploiting the relative frequencies of the various operations in large meshes, we have devised a slightly different code that works better in practice, but no longer guarantees never to exceed $2t$ bits. We encode CC, CS, and CR pairs as single symbols. We break the CLERS sequence into symbols of one or two letters each. Each symbol is one of the seven words listed in the table below. For each word, we suggest a binary code and indicate the number of letters that would be part of such words in a 100 letters sub-string generated for a typical model (we used the 69,674 triangle Stanford Bunny). The right column indicates the bit-cost associated with the occurrences of the word in a 100 letter sub-string.

For the models that we have explored, the resulting file size varies between $1.3t$ bits (for the Bunny model) and $1.6t$ bits (for a 2D Delaunay triangulation).

Word	code	# of letters	Cost
CR (after even Cs)	01	53.6	53.6 bits
CC (after even Cs)	00	22.4	22.4 bits
CS (after even Cs)	1101	1.2	2.4 bits
R (after even Cs)	10	19.6	39.2 bits
E	1100	1.6	6.4 bits
S (after even Cs)	1111	0.9	3.6 bits
L	1110	0.3	1.2 bits
TOTAL		100	128.8 bits

Custom 0.91t to 1.26t bit entropy codes

For large meshes, we construct a Huffman code as follows. We introduce a space after each non-C symbol that is followed by a C. The resulting words start with one or more C symbols, which are followed by one or more non-C symbols.

Experimenting with Delaunay triangulations of 200,000 triangles, we found only 1,400 words. A Huffman encoding of these words yields $1.26t$ bits. The table of codes takes about 32,000 bits ($0.16t$ bits for meshes with 200,00 triangles), but a part of it corresponding to most frequent words can be preloaded and kept constant for all large meshes.

For more realistic models (such as the 69,674 triangle Stanford Bunny) the dictionary had only 173 words and the cost of Huffman encoding is $0.85t$ bits. The total cost, including the dictionary is $0.91t$ bits.

Other general-purpose progressive coding techniques [23, 24, 12] may provide even better compression ratios for very large and regular meshes.

Comparison with general purpose techniques

We use the 69,674 triangle Stanford Bunny to illustrate the performance of the Edgebreaker compression. An ASCII file representing the uncompressed triangle table requires $133t$ bits. It may be compressed down to $51t$ bits with gzip. Encoding the vertex references as 16-bit binary integers, instead of ASCII characters, yields $48t$ bits, which may be further compressed to $39t$ bits with gzip.

With Edgebreaker's CLERS format, the connectivity may be encoded with $2.0t$ bits using the standard code and can be gzipped down to $1.16t$ bits. Our improved code of one and two letter symbols yields $1.3t$ bits and can be gzipped down to $0.98t$ bits. Our entropy code yields $0.85t$ bits and can be gzipped down to $0.83t$ bits.

In conclusion, Edgebreaker provides roughly a 50-to-1 compression ratio for gzipped files representing the connectivity of triangle meshes, which is usually the main storage factor for uncompressed representations. Further attempts to gzip the best CLERS format result in a less than 2% improvement.

Wrap&Zip decompression

The decompression algorithm receives a binary encoding of the CLERS string and reproduces a labeled planar triangle graph that is homeomorphic to the original graph and has its vertices labeled as discussed in the compression section. The process is very simple and has two phases: Wrapping and Zipping.

Wrapping a triangle spanning tree

The Wrap&Zip decompression starts with the two initial triangles that correspond to the two initial C operations that need not be explicitly encoded in the CLERS string. One of their external edges is identified as the gate. We read the string and for each operation, attach a new triangle to the gate. Depending on the next op-code in the CLERS string, we select zero, one or two of the free edges of the new triangle as the gates (Fig. 3). A stack is used to keep track of gates for the left branches of S operations, where the triangle-tree bifurcates. On an E operation, the current branch of the triangle tree terminates and we pop the stack to expose a new gate.

Orienting the free edges

Edges that have never been gates are called *bounding* edges. They have only one incident triangle in the triangle-tree. A triangle of type E has two bounding edges. S has none. C and L have a left bounding edge and R has a right bounding edge. During the construction of the triangle-tree, we orient these bounding edges as shown with thick blue arrows on Fig. 3.

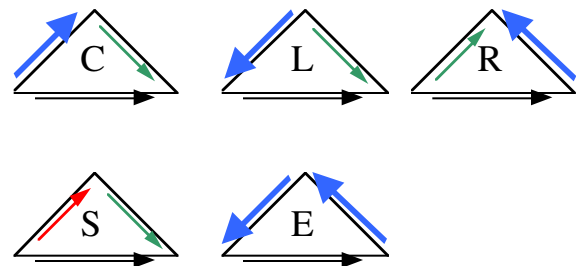


Figure 3: A new triangle is attached to the gate (black horizontal arrows). The new gates are indicated by green arrows. The red arrow (left gate of the S triangle) indicates a gate pushed onto the stack for the S operation. The thick blue arrows indicate the orientations of the bounding edges.

Zippering the wrap

Each time two adjacent bounding edges point towards their common vertex (i.e. their blue-arrow point towards that vertex), we zip them together and identifying their other vertices to have the same label. This zip operation may be applied recursively (see Fig. 4). Note that no zipping is possible for C, R, and S operations and that recursive zipping occurs only for E operations, starting from the left vertex (the starting vertex of the gate).

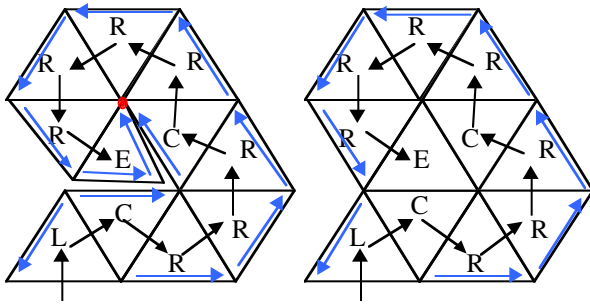


Figure 4: The black arrows show the sequence in which the triangles were constructed from the sub-string L C R R R C R R R R E. The blue arrows (left) indicate the bounding edges and their orientation prior to the zippering for the last triangle in the sequence. The zippering operation starts at the red vertex and zips two pairs of edges. The result is shown right.

Justification of the Wrap&Zip approach

The triangle-spanning tree produced by the Wrap&Zip technique is identical to the tree produced by several other approaches [5,17,20,13]. It may always be converted into the correct connectivity graph by gluing pairs of its bounding edges. In fact, the vertex-spanning tree of the Topological Surgery [17] is used to encode precisely this gluing information.

C operations are the only ones to add vertices to the vertex-spanning tree. If we orient the edges of the vertex spanning tree downwards (away from the root), then C operations create triangles that lie on the right of such bounding edges. R, E, and L operations create triangles that lie on the left of such bounding edges. Wrap&Zip zips up the vertex-spanning tree starting from its leaves, which are vertices with two incident bounding edges that point to them (as defined by the orientation of the blue arrows of Fig. 4, which is identical to the downward orientation of the edges of the vertex-spanning tree, away from the root).

Zippering an edge corresponds to removing it from the vertex-spanning tree. Our recursive procedure ensures that we zip up all the children before we zip up past a branching node of the vertex-spanning tree.

Time complexity

We start the recursive zippering procedure at most t times: once for each L and each E operation. Consequently, we stop the zippering procedure the same number of times. (The zippering procedure only goes up the vertex-spanning tree and does not

bifurcate.) We conclude that the number of times we test a vertex and decide not to zip it is bounded by t . The number of successful zip operations equals the number of edges in the vertex-spanning tree, which is precisely $v-1$. Therefore, the decompression algorithm has linear time complexity.

Implementation

Both the Edgebreaker compression and the Wrap&Zip decompression algorithms have been tested on a variety of meshes. Running on a single processor SGI Power Challenge, compressing the bunny model with 69,674 triangles took 3.87 seconds and decompression took 0.38 seconds. This decompression rate of 184K triangles per second was achieved without any attempt to optimize performance.

Our implementation of the Wrap&Zip decompression has 350 lines of C code and proceeds as follows.

Reading the CLERS string, we build the triangle table with vertex IDs. The vertex IDs correspond to a systematic labeling of the vertices as first encountered in the triangle tree. Some of these references will be updated after the zippering phase. We also build a vertex table, which stores for each vertex:

- pointers to the previous and next vertices in the bounding loop of P (the topological polygon represented by the triangle tree)
- the orientations of the two incident edges in that loop
- two pointers for building cycles of coincident vertices, as identified during the zip

We traverse the bounding loop and zip recursively as appropriate. At each zipped pair of edges, we update the doubly linked list of references to the bounding vertices and merge the cycles of the two coincident vertices.

Finally, we allocate a unique vertex label for each cycle and update the triangle table.

More general meshes

Boundaries of non-manifold solids may be represented using topological graphs of manifold solids that treat non-manifold vertices and edges as if they were split into distinct manifold elements, which happen to coincide in space. Such representations are sometimes called pseudo-manifolds.

The boundary of a manifold or pseudo-manifold solid may be composed of one or more connected components, which either bound separate solids or bound cavities in them. Each component is compressed and decompressed independently.

The Edgebreaker compression and the Wrap&Zip decompression algorithms, as described above, are limited to components without handles or holes. In this section, we first present a new extension of the compression and decompression algorithms that supports handles. Then we explain how to extend our Wrap&Zip decompression to support the approach originally proposed in [13] for handling holes (i.e., bounding loops in the surface).

To clarify the distinction between handles, holes, and cavities, consider that a torus has one handle (also, called through-hole), while a solid ball with an empty core has an enclosed cavity,

but no hole or handle (using our terminology). A spherical surface, from which one has cut out a disk, is no longer a closed boundary that separates space into disjoint components. We say that such a surface has a hole and is thus a two-manifold with boundary, having for boundary a single one-manifold curve. Of course, a surface may have multiple holes.

Handles

For clarity, we first describe how Wrap&Zip uses an extended CLERS code to reconstruct the connectivity of meshes with handles. Then, we suggest a format for storing this extended code. Finally, we explain how the extended code is generated by a modified version of Edgebreaker's compression algorithm.

The extended decompression algorithm reads the sequence of op-codes in the CLERS string and builds the triangle tree. However, it now must handle 6 types of triangles: the five C, L, E, R, and S types described above, plus the new S* type.

As for S triangles, the triangle-tree is grown further from the right edges of the S* triangles, but not from their left edges, which temporarily become bounding edges of the topological polygon P defined by the completed triangle-tree. The extended CLERS string associates with each S* triangle an integer identifying a matching edge that also bounds P.

Once the entire triangle-tree is built, Wrap&Zip traverses the boundary of P and builds an array of edge-pointers indexed by the integer edge ID incremented as we visit the consecutive edges along the bounding loop. This array will speed up the identification of the matching edges for each S* triangle.

Then, Wrap&Zip traverses the triangle-tree again and glues the left edge of each S* triangle with the edge identified by the integer stored with the triangle. (That integer is used as an index into the array of edge-pointers.) Each glue operation stitches the mesh, merging two edges into one and merging their four vertices into two. Note that for orientable surfaces, there is no ambiguity as to the relative orientation of the two edges being glued.

Note that changing the order of the gluing operations does not affect the topology of the final result because the matching edges are identified independently of each other. Also note that each handle in the original mesh results in two S* triangles (see [17]). The gluing operation associated with one of them will split the boundary of P into two disjoint loops. The second one will merge these two loops back into a single loop because the left edge of its S* triangle and the edge that it should be glued to are in separate loops. Consequently, executing the $2h$ gluing operations, where h is the number of handles, restores a single boundary for P. The resulting topological model represents a shape which no longer is a simple polygon, but is topologically equivalent to a surface obtained by cutting a small disk out of the original surface. The boundary of that disk is a single loop of edges. Each edge in the loop coincides physically with another edge of the loop. Wrap&Zip does not need to identify this correspondence through global gluing operations nor through geometric coincidence tests. Instead, it applies the "zipping" process described earlier and restores the original manifold or pseudo-

manifold surface with handles by a series of local zipping operations that glue pairs of adjacent edges that point towards their common vertex (using the orientation of the free edges derived from the op-code associated with each triangle).

The generalization of our CLERS format must contain the information necessary to identify the S* operations and the integer edge-identifier associated with each S* triangle. One could add one bit to the Edgebreaker code used for the S triangles in order to distinguish S triangles from S* triangles. When the number of handles, h , is small compared to the number of S triangles, it is more compact to use the same code for S and S* operations and to distinguish them by storing a table of integer counts. Each count indicates how many of these S-or-S* triangles that precede the current S* triangle were actually of type S. This count could be the total number of S triangles preceding the current S* or just their count from the previous S* (or from the beginning for the first S* in the CLERS code). With each count in the table we also associate the edge identifier.

There are $t+2$ free edges bounding P. Therefore we need $2h \log_2(t+2)$ bits to store all the edge identifiers. If the table is decoded after the CLERS string, we know s , the total count of S or S* triangles, and need only $2h \log_2(s)$ bits to identify the $2h$ triangles of type S*. In practice, s is about $t/20$ or less. The number of handles varies from model to model, but is typically much smaller than s .

Let us now describe how Edgebreaker's compression algorithm may be adapted to produce the table for handles when generating the modified CLERS code for supporting meshes with handles.

Compression proceeds as before labeling the encountered triangles as C, L, E, R, or S. Some of the S triangles are temporarily mislabeled during this process and will be turned later into S* triangles.

For simplicity, and without loss of generality, we assume that the mesh is represented using some variation of a half-edge structure. A half-edge is an abstract entity associating a triangle with one of its bounding edges. The half-edge has a natural orientation defined by the outward-pointing normal to the triangle (see discussion in [13]). Each half edge is associated with the next and previous half-edge around its triangle and with the opposite half-edge, which associates another triangle with the same edge. The construction of the triangle-spanning tree may be easily performed by following these half-edge pointers. The boundary of the topological region P defined by the triangle-tree is represented by an ordered list of pointers to bounding half-edges (see [13]).

If we also store with each half-edge a pointer to its associated triangle and to its starting vertex, we can easily recover and manipulate the vertex and triangle markings used by Edgebreaker. Finally, given a half-edge, we can recover the opposite triangle.

When compression reaches an E triangle, it pops out a new half-edge pointer from the stack of left-edges of previously encountered S triangles. Let X denote the triangle associated

with it and let O denote the opposite triangle. If O has not been visited, then X is a genuine triangle of type S and a new branch of the triangle-tree will be constructed starting with O . If however, the opposite triangle is marked as already visited, then X is simply relabeled as a triangle of type S^* .

Once all the triangles are labeled, we traverse the loop of bounding half-edges and number them using increasing integers. The ID is initialized to zero and we start the process at the first bounding half-edge of the first triangle of the triangle-tree. Each time we encounter a half-edge that has an opposite triangle of type S^* , we store with that triangle the ID of the half-edge.

Finally, we traverse the triangle-tree a second time exploiting the triangle labels to avoid the initial tests that define the traversal. We keep track of the counts of S and S^* triangles; store the triangle labels in the CLERS string (replacing each S^* with an S); and produce the appropriate entries into our table, which identifies the S^* triangles and the associated glue-edges.

Holes

Holes are processed as follows. The vertices in each hole form an ordered cyclic list. We label the vertices of the mesh in the following order. Each time a C triangle is processed, we label its tip vertex with the current value of the counter and increment the counter. Each time we encounter a hole for the first time, we visit its vertices in their cyclic order, starting with the vertex at which we have touched the hole's boundary. We label these vertices incrementing our counter for each and we mark them as visited. Furthermore, we label the current triangle as S' and associate with it the number of vertices in the boundary of the newly discovered hole.

Then, we encode each S' triangle using a table similar to the table used for S^* triangles. S' triangles are stored as S triangles in the CLERS string. The S' table contains the count of S symbols that separate the current S' from the previous one in the string or from the beginning of the string. It also contains the number of vertices in the corresponding hole.

At decompression, Wrap&Zip uses the table to distinguish S' from S and from S^* triangles. Each time an S' triangle is decoded, Wrap&Zip appends the corresponding number of edges and vertices to the current loop at the tip of the S' triangle.

Conclusion

The Wrap&Zip technique reported here provides a simple and efficient decompression algorithm for the connectivity of 3D triangle meshes that have been compressed with the Edgebreaker approach and are represented in the CLERS format. For meshes homeomorphic to a sphere, the decompression algorithm has linear time and space complexity and our simple 350 lines implementation decompresses about 184K triangles a second.

By analyzing the statistics of the op-codes generated by the compression process for a variety of models, we have developed a model-independent coding scheme, which

compresses the connectivity of t triangles to between $1.3t$ and $1.6t$ bits. This cost may be further reduced for large models down to $0.85t$ and $1.29t$ bits using entropy codes.

A variation of this approach has been used by the authors to compress the connectivity graph of tetrahedral meshes [16].

In addition to the new Wrap&Zip decompression algorithm, we have provided a simpler description of the Edgebreaker compression process and have introduced a new approach for extending these algorithms and the associated CLERS format to meshes with handles and holes.

Acknowledgement

Rossignac's contribution to this project was supported by NSF grant 9721358. Szymczak's work was supported by KBN grant 0449/P3/94/06.

Bibliography

- [1] R. Chuang, A. Garg, X. He, M. Kao, and H. Lu, Compact Encoding of Planar Graphs via Canonical Orderings and Multiple Parentheses, in *Automata, Languages, and Programming*, Lecture Notes in Computer Science 1443, Springer, Eds. K. Larsen, S. Skyum, and G. Winskel, pp. 118-129, Proc. 25th International Colloquium, ICALP'98, Denmark, July 1998.
- [2] M. Deering, Geometry Compression, Computer Graphics, Proceedings Siggraph'95, 13-20, August 1995.
- [3] M. Denny and C. Sohler, Encoding a triangulation as a permutation of its point set, Proc. of the Ninth Canadian Conference on Computational Geometry, pp. 39-43, Ontario, August 11-14, 1997.
- [4] F. Evans, S. Skiena, and A. Varshney, Optimizing Triangle Strips for Fast Rendering, Proceedings, IEEE Visualization'96, pp. 319--326, 1996.
- [5] S. Gumhold and W. Strasser, Real Time Compression of Triangle Mesh Connectivity. Proc. ACM Siggraph 98, pp. 133-140, July 1998.
- [6] X. He and M. Y. Kao and H. I. Lu, Linear-Time Succinct Encodings of Planar Graphs via Canonical Orderings, to appear in the SIAM Journal on Discrete Mathematics, 1999.
- [7] A. Itai and M. Rodeh, Representation of Graphs, Acta Informatica, No. 17, pp. 215-219. 1982.
- [8] K. Keeler and J. Westbrook, Short Encodings of Planar Graphs and Maps, Discrete Applied Mathematics, No. 58, pp. 239-252, 1995.
- [9] D. Kirkpatrick, Optimal search in planar subdivisions, SIAM Journal on Computing, vol 12, pp. :28-35, 1983.
- [10] D.T. Lee and F.P. Preparata, Location of a point in a planar subdivision and its applications. SIAM J. on Computers, 6:594-606, 1977.
- [11] M. Naor, Succinct representation of general unlabeled graphs, Discrete Applied Mathematics, vol. 29, pp. 303-307, North Holland, 1990.
- [12] M. R. Nelson, LZW Data Compression, Dr. Dobb's Journal, October 1989.
- [13] J. Rossignac, Edgebreaker: Compressing the incidence graph of triangle meshes, IEEE Transactions on Visualization and Computer Graphics, Vol. 5, No. 1, January - March 1999.

- [14] J. Rossignac, 3D Geometry Compression: Just-in-time upgrades for triangle meshes, in *3D Geometry Compression*, Course Notes 21, Siggraph 98, Orlando, Florida, July 18-24, 1998.
- [15] J. Snoeyink and M. van Kerveld, Good orders for incremental (re)construction, Proc. ACM Symposium on Computational Geometry, pp. 400-402, Nice, France, June 1997.
- [16] A. Szymczak and J. Rossignac, Grow&Fold: Compression of Tetrahedral Meshes, ACM Symposium on Solid Modeling, 1999.
- [17] G. Taubin and J. Rossignac, Geometric Compression through Topological Surgery, ACM Transactions on Graphics, Volume 17, Number 2, pp. 84-115, April 1998.
- [18] G. Taubin, W. Horn, F. Lazarus, and J. Rossignac, Geometry Coding and VRML, Proceedings of the IEEE, pp. 1228-1243, vol. 96, no. 6, June 1998.
- [19] G. Taubin and J. Rossignac, *3D Geometry Compression*, Course Notes 21, Siggraph 98, Orlando, Florida, July 18-24, 1998.
- [20] C. Touma and C. Gotsman, Triangle Mesh Compression, Proceedings Graphics Interface 98, pp. 26-34, 1998.
- [21] G. Turan, Succinct representations of graphs, Discrete Applied Math, 8: 289-294, 1984.
- [22] W.T. Tutte, The Enumerative Theory of Planar Graphs. In *A Survey of Computational Theory*, J.N. Srinivasan et al. (Eds.). North-Holland, 1973.
- [23] T. Welch, A Technique for High-Performance Data Compression, Computer, June 1984.
- [24] J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, May 1977.