Wrap&Zip decompression of the connectivity of triangle meshes compressed with Edgebreaker☆

Jarek Rossignac and Andrzej Szymczak

GVU Center, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280

Abstract

The Edgebreaker compression [16, 11] is guaranteed to encode any unlabeled triangulated planar graph of t triangles with at most 1.84t bits. It stores the graph as a CLERS string¾a sequence of t symbols from the set {C,L,E,R,S}, each represented by a 1, 2 or 3 bit code. We show here that, in practice, the string can be further compressed to between 0.91t and 1.26t bits using an entropy code. These results improve over the 2.3t bits code proposed by Keeler and Westbrook [10] and over the various 3D triangle mesh compression techniques published recently [7,9,14,24,25], which exhibit either larger constants or cannot guarantee a linear worst case storage complexity. The decompression proposed in [16] is complicated and exhibits a non-linear time complexity. The main contribution reported here is a simpler and efficient decompression algorithm, called Wrap&Zip, which has a linear time and space complexity. Wrap&Zip reads the CLERS string and builds a triangle tree¾a triangulated, simply connected, topological polygon without interior vertices. During that process, it uses a simple rule to orient the external edges of this polygon and to "zip" (i.e., identify) all pairs of adjacent external edges that are oriented towards their common vertex. Because triangulated planar graphs can only model the connectivity (triangle/vertex incidence) of 3D triangle meshes that are homeomorphic to a sphere, we introduce here simple extensions of the Edgebreaker and Wrap&zip techniques for compressing more general, manifold, triangle meshes with holes and handles. Manifold representations of non-manifold meshes are discussed in [18,5,6]. The simple and efficient solution provided by the combination of Wrap&Zip with of the work reported in [16, 11] yields a simple and effective solution for compressing the connectivity information of large and small triangle meshes that must be downloaded over the Internet. The CLERS stream may be interleaved with an encoding of the vertex coordinates and photometric attributes enabling inline decompression. The availability of local incidence information permits to use, during decompression, the location and attributes of neighboring vertices to predict new ones, and thus supports most of the recently proposed vertex compression techniques [2,7,21,24].

Keywords: Triangle graphs, Triangle Mesh, Encoding, Compression, Connectivity

1 Introduction

1.1 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).

☆ This material is based upon work supported by the National Science Foundation under Grant 9721358 and by KBN under grant 0449/P3/94/06.

1.2 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 "$\lceil \rceil$" 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 [25] 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 [26], Itai and Rodeh [9] 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 [7, 16, 17, 21, 22].

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 [21, 22] 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 [17] 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 [7] and Rossignac's Edgebreaker scheme [16], although developed independently, are closely related. Both schemes perform the same traversal of the mesh as in [21]. 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 [10] 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 [24] also encode the vertices along the vertex-spanning tree in the same spiraling order as [7,16,21]. 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 [7,9,24] is illustrated in Fig. 1. The triangulated region shows only the first four levels of a recursive construction of a portion of a simple mesh that contains v vertices and would be encoded with only S and E symbols using Edgebreaker. The mesh can be made arbitrarily more complicated by unfolding the recursion further. Such a situation may indeed occur in a simple mesh. For example, consider using this region as the basis of a cone, adding v triangles that are all incident upon a single new vertex (not shown here).

Encoding this mesh with techniques proposed in [7,9,24] without using Huffman codes would require: log(v/2)+2log(v/4)+4log(v/8)+8log(v/16)…(v/4)log(2) bits. Huffman codes, used in [7,24], may reduce the cost in this regular example to O(vloglog(v)).

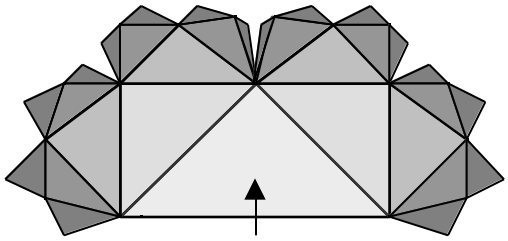

Figure 1: A pathological case for which the storage of offsets is non-linear.

Inspired by [12] and improving on [14, 19], 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 [13] 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. [1,8] 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.

2 The Edgebreaker compression

The Edgebreaker compression algorithm was introduced in [16]. We include it here for completeness and attempt to explain it in a simpler and more concise manner, which may help readers to develop more effective implementations.

2.1 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 could be built by a recursive procedure, which visits adjacent triangles and marks them. The procedure starts with any triangle x of the mesh and selects one of its edges, e, which (assuming a convention for orienting the triangles) suffices to define x.right and x.left, the other two triangles adjacent to x but not incident upon e. This initial triangle will be the root of the triangle tree. Then recursion proceeds as follows. If x.right has not been previously visited, it is appended to the tree as the right child of x and is visited by the recursive procedure in a depth-first order. Then, if x.left has still not been visited, it is appended as the left child of x and is also visited by the recursion. During the process, we mark and number all the vertices of visited triangles.

There is really no need to build the triangle-tree. The encoding process may be implemented as a state machine. Let P denote the parent of x in the triangle tree and let **v** be the only vertex of x that is not the vertex of P. The "visited or not-visited" status of **v**, of x.left, and of x.right unambiguously identify one out of the five possible situations depicted in Fig. 2 and associated with the letters: C, L, E, R, and S. A stack is used to save the left neighbors of S triangles until the corresponding E triangle is encountered.

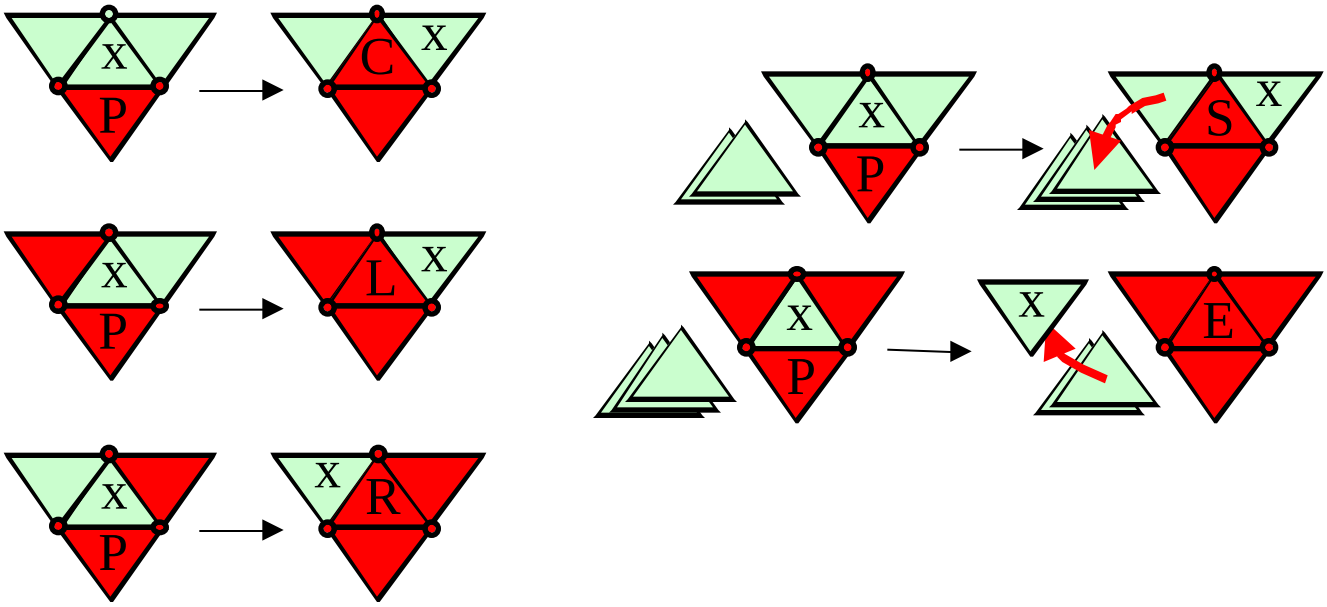

Figure 2: Edgebreaker's compression state machine

The following test illustrates the simple decision process:

IF NOT ***v****.marked THEN C*
ELSE IF x.left.marked
THEN IF x.right.marked THEN E ELSE L
ELSE IF x.righ.marked THEN R ELSE 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.

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 [16].

The early stages of the algorithm are illustrated in Figure 3. Starting from the darker central triangle with his horizontal edge e identified by a thick line, Edgebreaker visits the triangles along a spiral and stores the following labels at the beginning of the CLERS sequence: CCCCRCCRCRC.

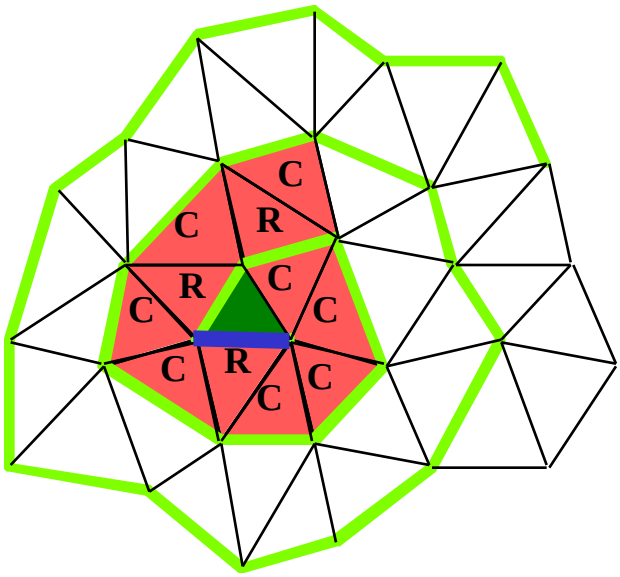

Figure 3: A typical starting phase of Edgebreaker's compression

Figure 4 illustrates the ending of a compression process. The empty triangles that have not yet been visited (left) are surrounded by previously visited triangles (not shown here). Edgebreaker visits them and produces the sequence CRSRLECRRRLE, as shown (right).

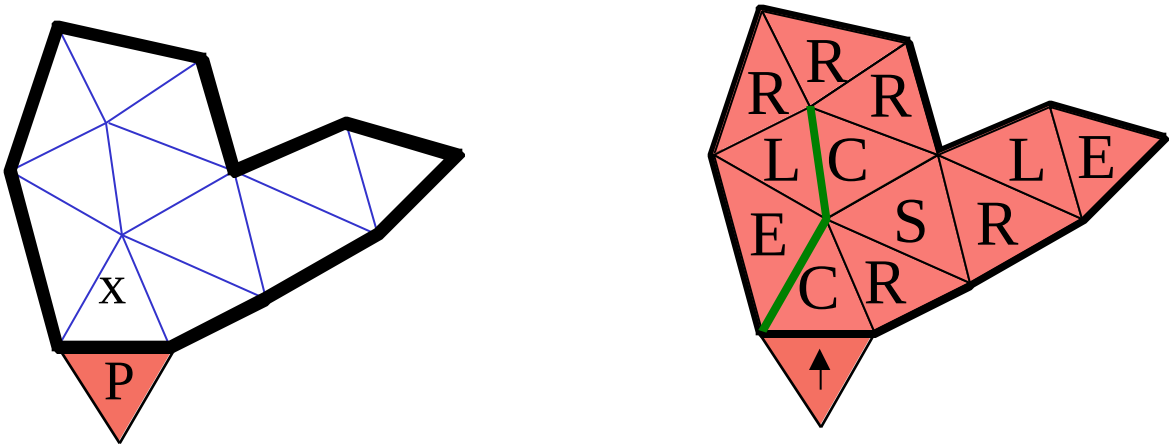

Figure 4: A typical ending of Edgebreaker's compression

3 Guaranteed bound encoding

3.1 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 $2t$ bits.

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

3.2 Expected 1.7t bit code not exceeding 2t bits

CL and CE combinations are impossible, because a **v** vertex of a C triangle must have been unmarked to enable the C operation and must have been marked to enable the subsequent L operation. Consequently, 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 2*t* bits.

3.3 Guaranteed 1.84t bit code

A slightly more complex code, developed by King and Rossignac in [11], is guaranteed to store any valid CLERS sequence in less than 1.84*t* bits. It encodes C as 0, S as 10 and R as 11, when they do not follow a C. Symbols that follow a C are encoded using one of three possible codes. Code I: C is 0, S is 100, R is 101, L is 110, E is 111. Code II: C is 00, S is 111, R is 10, L is 110, E is 01. Code III: C is 00, S is 010, R is 011, L is 10 and E is 11.

In [11], it is proven that, for any given simple mesh, one of these three codes is guaranteed to yield a total storage of less than (1+5/6)*t* bits. We simply produce all three in parallel and store the shortest, preceded by a 2-bit identifier that will tell the decoder which code was used for the particular mesh.

4 Practical results

The compression results reported here refer to the extended encoding, discussed later in this paper, which supports meshes with handles and holes.

4.1 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 2*t* 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.

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

We have explored a variety of models. The compressed file size varies between 1.3*t* bits (for a typical models such as the Stanford Bunny) and 1.6*t* bits (for 2D Delaunay triangulations).

4.2 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 1.26*t* bits. The table of codes takes about 32,000 bits (0.16*t* 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 0.85*t* bits. The total cost, including the dictionary is 0.91*t* bits.

Other general-purpose progressive coding techniques [27, 28, 15] may provide even better compression rations for very large and regular meshes.

4.3 Comparison with general purpose techniques

We use the 69,674 triangle Stanford Bunny to compare the performance of the Edgebreaker compression with respect to other general purpose compression techniques.

An ASCII file representing the uncompressed triangle table requires 133*t* bits. It may be compressed down to 51*t* bits with gzip. Encoding the vertex references as 16-bit binary integers, instead of ASCII characters, yields 48*t* bits, which may be further compressed to 39*t* bits with gzip. With Edgebreaker's CLERS format, the connectivity may be encoded with 2.0*t* bits using the standard code and can be gzipped down to 1.16*t* bits. Our improved code of one and two letter symbols yields 1.3*t* bits and can be gzipped down to 0.98*t* bits. Our entropy code yields 0.85*t* bits and can be gzipped down to 0.83*t* 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.

5 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, which may be performed sequentially or simultaneously. Furthermore, if inline decompression is desired, so that the receiver may build (and possibly render) the initial portions of the mesh before receiving the rest, the encoding of the vertices may be interleaved with the CLERS sequence.

5.1 Wrapping a triangle spanning tree

The Wrap&Zip decompression starts by initializing the mesh with two triangles, P and x. The right edge of x, when coming from P, is identified as the gate. We read the CLERS string and for each operation, attach a new triangle to the gate. Depending on the symbol associated with the current triangle, we select zero, one or two of the free edges of the new triangle as the gates (Fig. 5). A stack is used to keep track of gates for the left branches of S operations. On an E operation, we pop the stack to expose a new gate. This process follows the same sequence as the encoding process of Figure 2, but creates the triangles and vertices, rather than marking them. At each C operation, a new vertex is created and, if we use inline codes, its encoding is decoded from the input stream. At each S operation, a new dummy vertex-reference is created. It will be replaced later by a reference to a previously decoded vertex.

5.2 Orienting the free edges

The above process builds a simply connected triangulated manifold surface with boundary. Its boundary is a simply connected loop of bounding edges that each have a single incident triangle. One of the bounding edges is the gate. That is where we will attach the next triangle. The other bounding edges are oriented as shown in Fig. 5. When a new triangle is attached to a previously reconstructed triangle P, its free edges that are not gates are oriented clockwise, except for C-type triangles.

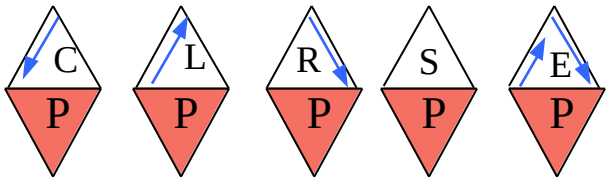

Figure 5: Orientations for the free edges of the polygon produced by the Wrap&Zip decompression

Note that, except for C triangles, all bounding edges are oriented clock-wise. Each bounding edge of a C triangle will be matched during the zipping operation, described below, with a bounding edge of a non-C triangle. The Wrap&Zip solution presented here is based upon the discovery that this bounding edge orientation suffices to unambiguously recover the original match, which may be reconstructed by applying the zipping rule (defined below).

5.3 Zipping the wrap

Each time two adjacent bounding edges point away from their common vertex, we zip them together by identifying their other vertices with the same label. This zip operation is applied recursively (see Fig. 6) as long as the two edges incident upon the newly merged vertex point away from it. We use the analogy of a zipper. We place the zipper at vertices, from which two arrows depart and keep zipping in the direction of the arrows, as long as the arrows on the left and right side of the crack agree.

Note that no zipping may be initiated by the creation of a C triangle, because the matching edge has not yet been created. Also, note that no zipping may be initiated by the creation of an R triangle, because its arrow is pointing towards a vertex that bounds the gate (which is not oriented). Finally, S triangles have no arrows, since both their free edges will be used as gates.

Figure 6 (a) shows a hole in the mesh, which has to be filled with triangles. The bounding edges of previously recovered triangles have been oriented during their construction, as indicated by the arrows. The gate is marked by a thicker edge. Wrap&Zip will decode the *CRSRLECRRRLE sequence of symbols and reconstruct the mesh that was encoded by Edgebreaker in Figure 4 and zip it with the previously decoded mesh. No zipping occurs during the decoding of the* CRSRLECRRR subsequence.

Figure 6 (b) shows the result of decoding that CRSRLECRRR sub-sequence and the following L triangle, just before the first zipping operation occurs. The bounding edge of the last L- triangle and the corresponding bounding edge of the C-triangle have compatible orientations and will be unified. The new gate is marked with a thicker line.

Figure 6 (c) shows the result of the unification for the first zipping step, followed by the creation of an E-triangle, which corresponds to the last symbol in the CLERS string. The merged vertex is marked by a large dot. A new zipping operation will start at that vertex and first zip the C and E triangle, then continue to zip the new region with the initial mesh of Figure 6 (a). The zip follows the direction of the arrows until all bounding edges are matched. The result is shown Figure 6 (d).

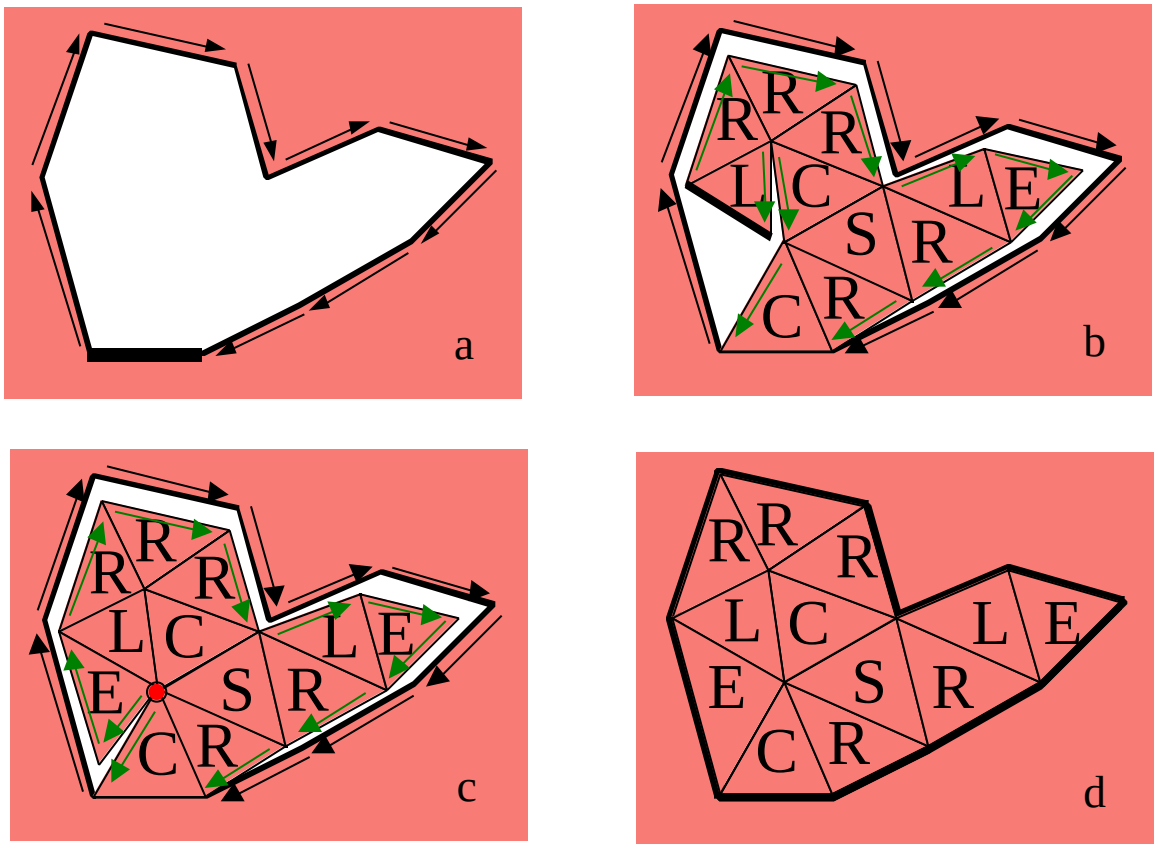

Figure 6: Wrapping and zipping CRSRLECRRRLE.

5.4 Justification of the Wrap&Zip approach

Without zipping, Wrap&Zip would produce a triangulated polygon that corresponds to the triangle-spanning trees encoded in the Topological Surgery compression technique [21]. In addition to the triangle-spanning tree, the Topological Surgery decompression uses an encoding of a vertex-spanning tree to recover the information necessary to match the bounding edges of that polygon. (The vertex-spanning tree is the dual of the triangle-spanning tree and is formed by the edges of the original mesh that are not edges of the triangle-spanning tree.)

C operations are the only ones to add vertices to the vertex-spanning tree. If we orient the edges of the vertex-spanning tree upward (towards the root), then C operations create triangles that lie on the left of such bounding edges. R, E, and L operations create triangles that lie on the right of such bounding edges. The zipping follows the vertex-spanning tree starting from its leaves, which are vertices with two incident bounding edges that point away from them (as defined by the orientation of the arrows). That orientation is identical to the upward orientation of the edges of the vertex-spanning tree.

Zipping 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.

5.5 Time complexity

We start the recursive zipping procedure at most t times: once for each L and E operation. Consequently, we stop the zipping procedure the same number of times. (The zipping 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.

5.6 Implementation

Both the Edgebreaker compression and the Wrap&Zip decompression algorithms have been tested on a variety of meshes. For example, 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 zipping phase. We also build a vertex table, which maintains, for each vertex:

- The Next_vertex and Previous_vertex pointers to the previous and next vertices along the bounding loop of the mesh that has been recovered and zipped so far,
- The Left_edge and Right_edge Booleans indicating the orientations of the two incident bounding edges and the Left-Gate and Right_Gate Booleans indicating whether the incident edges are the active or a stacked gate,
- Next_Same_Vertex and Previous_Same_Vertex pointers to a doubly-linked cyclic list of the set of coincident vertices, as identified during the zip.

As we add triangles to the current mesh, we update the Next_vertex and Previous_vertex pointers to maintain the bounding loop. We also set the Booleans as needed.

When an L or E triangle is created, we check their “left” vertex. If its Left_Edge and Right_Edge are both TRUE (i.e., the edges point away from the vertex), we start a recursive zipping process.

As each edge-pair is zipped, we update the bounding loop by removing two vertices from the doubly linked list and by adjusting the Next_Vertex and Previous_Vertex pointers of their neighbors. We also merge the cycles of the two vertices that were identified by the zip.

At the end of the zipping process, we allocate a unique vertex label to each cycle and update the triangle table to reflect this unique vertex labeling. The labels are increasing integers in the order in which the corresponding vertices are first encountered by C operations.

When compressing 3D meshes, the vertices are encoded as the corrective vectors between their quantized location and a predictive location derived from neighboring vertices that have lower integer labels. These encodings may be interleaved in the CLERS code to immediately follow the corresponding C symbol.

6 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.

Pseudo-manifold representations of non-manifold triangulated solids may be produced with nearly minimal vertex replication using techniques introduced in [18]. Techniques for compressing more general, non-manifold meshes are discussed in [5,6].

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

The simple versions of 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 [16] 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.

6.1 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 from the right edges of the S* triangles, but not from their left edges, which temporarily become bounding edges. The extended CLERS string associates, with each S* triangle, an integer identifying a matching bounding edge that is glued to the left edge of the S triangle prior to zipping.

Figure 7 shows a typical process of encoding a handle. Before it is processed by the compression algorithm, a handle (top-left) has a simply connected bounding loop. A fist S* triangle breaks the mesh into a topological polygon with one hole (top-right). A second S* triangle unifies the outer loops with the hole-bounding loop (bottom left). With each S* triangle is associated a reference to the corresponding edge (bottom-right).

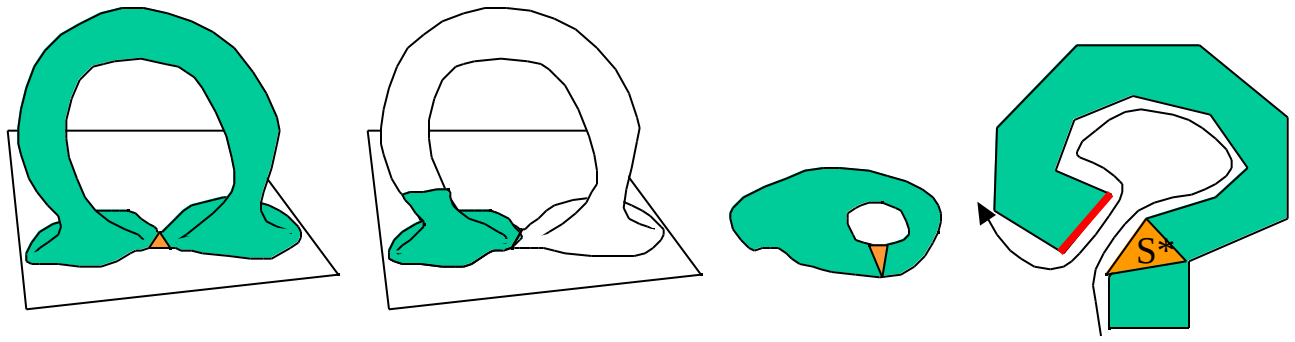

Figure 7: Each handle corresponds to two S* triangles

Wrap&Zip decodes the CLERS sequence as described previously, but marking the left edges of S*-triangles as glue-edges. It builds the triangulated polygon and performs zipping operations whenever possible.

Once the entire triangle-tree topological polygon is built, Wrap&Zip traverses its boundary 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 the glue-edges of 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 an integer associated with the triangle. (That integer is computed during compression and stored in a separate table. It 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.

Also, note that changing the order of the gluing operations does not affect the topology of the result because the matching edges are identified independently of each other.

Furthermore, note that each handle in the original mesh produces two S* triangles (see [21]). 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 (see Fig. 7). Consequently, executing the $2h$ gluing operations, where h is the number of handles, restores a single bounding loop.

The topological model that results from the gluing operations represents a shape, which no longer is a simple polygon. Instead, it 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 the 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 much smaller than 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 mush smaller then 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 [16]). 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 [16]). 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.

6.2 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. As before, 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 for the first S'). 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. The compression process is illustrated in Figure 8. It produces the string SRLRLRLRLRE and a hole table with a single entry: (1,5) indicating that the first S is an S' and that it connects to a hole with 5 vertices.

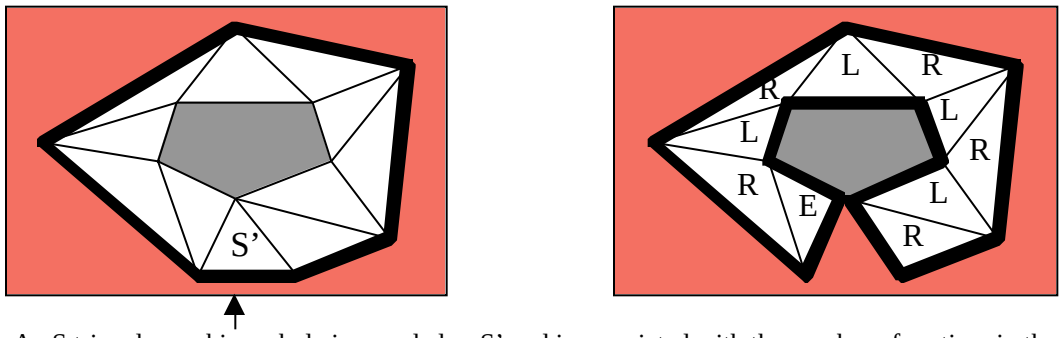

Fig. 8: An S triangle reaching a hole is encoded as S' and is associated with the number of vertices in the hole

Wrap&zip interprets the resulting "1 (1,5) SRLRLRLRLRE" encoding as follows. The first 1 indicates that there is a single S'. The 1 in the parenthesis indicates that this first S is an S'. The following 5 indicates that the associated hole has five vertices.

As shown in Fig. 9, Wrap&Zip appends an S' triangle to the gate (Fig. 9 left) and merges the resulting boundary with a loop of the five vertices of the hole (Fig. 9 center). Then it proceeds normally to fill in the missing triangles.

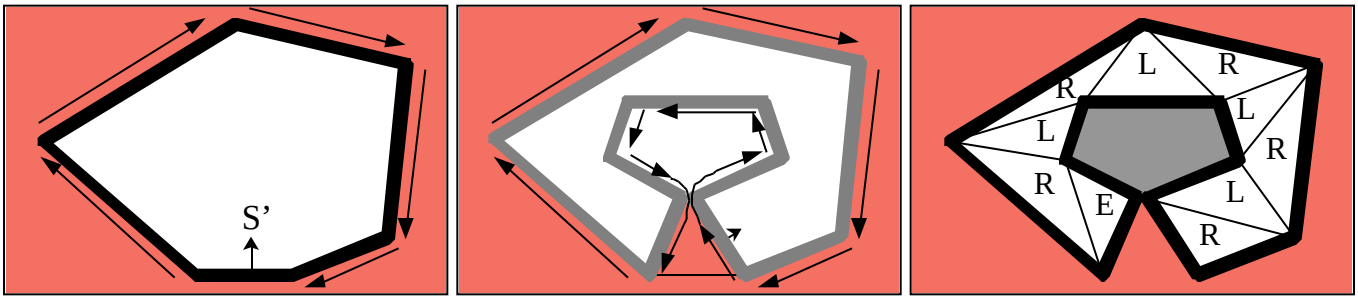

Figure 9: An S' is an S triangle that reaches a hole

7 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 three-dimensional version of this approach has been used by the authors to compress the connectivity graph of tetrahedral meshes [20].

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.

8 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, Danmark, 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 Vizualization'96, pp. 319--326, 1996.

[5] A. Gueziec, G. Taubin, F. Lazarus, and W. Horn, Converting sets of polygons to manifold surfaces by cutting and stitching. In Proc. IEEE Visualization' 98, pp. 383-390, October, 1998.

[6] A. Gueziec, F. Bosen, G. Taubin, and C. Silva, Efficient compression of non-manifold meshes, in Course Notes 22 on Geometric Compression, ACM Siggraph, August 99.

[7] S. Gumhold and W. Strasser, Real Time Compression of Triangle Mesh Connectivity. Proc. ACM Siggraph 98, pp. 133-140, July 1998.

[8] 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.

[9] A. Itai and M. Rodeh, Representation of Graphs, Acta Informatica, No. 17, pp. 215-219. 1982.

[10] K. Keeler and J. Westbrook, Short Encodings of Planar Graphs and Maps, Discrete Applied Mathematics, No. 58, pp. 239-252, 1995.

[11] D. King and J. Rossignac, Guaranteed 3.67v bit encoding of planar triangle graphs, In Proc. 11th Canadian Conf. On Computational Geometry, Vancouvert, August 1999.

[12] D. Kirkpatrick, Optimal search in planar subdivisions, SIAM Journal on Computing, vol **12**, pp. :28-35, 1983.

[13] 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.

[14] M. Naor, Succinct representation of general unlabeled graphs, Discrete Applied Mathematics, vol. 29, pp. 303-307, North Holland, 1990.

[15] M. R. Nelson, LZW Data Compression, Dr. Dobb's Journal, October 1989.

[16] J. Rossignac, Edgebreaker: Compressing the incidence graph of triangle meshes, IEEE Transactions on Visualization and Computer Graphics, Vol. 5, No. 1, January - March 1999.

[17] 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.

[18] J. Rossignac and D. Cardoze, Matchmaker, Manifold BReps for non-manifold R-sets. Proceedings of the ACM Symposium on Solid Modeling, 1999.

[19] 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.

[20] A. Szymczak and J. Rossignac, Grow&Fold: Compression of Tetrahedral Meshes, ACM Symposium on Solid Modeling, 1999.

[21] G. Taubin and J. Rossignac, Geometric Compression through Topological Surgery, ACM Transactions on Graphics, Volume 17, Number 2, pp. 84-115, April 1998.

[22] 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.

[23] G. Taubin and J. Rossignac, *3D Geometry Compression,* Course Notes 21, Siggraph 98, Orlando, Florida, July 18-24, 1998.

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

[25] G. Turan, Succinct representations of graphs, Discrete Applied Math, 8: 289-294, 1984.

[26] W.T. Tutte, The Enumerative Theory of Planar Graphs. In *A Survey of Computational Theory,* J.N. Srinivasan et al. (Eds.). North-Holland, 1973.

[27] T. Welch, A Technique for High-Performance Data Compression, Computer, June 1984.

[28] J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, May 1977.