Mesh Collapse Compression

by

Martin Isenburg

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

October 1999

Abstract

Efficiently encoding the topology of triangular meshes has recently been the subject of intense study and many representations have been proposed. The sudden interest in this area is fueled by the emerging demand for transmitting 3D data sets over the Internet (e.g. VRML). Since transmission bandwidth is a scarce resource, compact encodings for 3D models are of great advantage. In this work we present novel algorithms for encoding the topology of triangular and quadrilateral meshes. Our encoding algorithms are based on the edge contract operation, which has been used extensively in the area of mesh simplification, but not for efficient mesh topology compression. Furthermore we present a simpler decoding algorithm for Edgebreaker encoded triangle meshes.

Contents

List of Tables

List of Figures

Acknowledgements

I thank my supervisor Jack Snoeyink for a list of things that is much too long to fit here. I thank the International Computer Science Institute (ICSI) at Berkeley for the research opportunity during the summer of 1998. I thank Cafe Strada for their great Latte. I thank Gene Lee for his RASP tools and his technical support during the making of the mc-video. I thank Davis King for providing several quadrilateral meshes and for explaining the details of Edgebreaker during our *pool session* at SCG 99, Miami Beach. I thank Jarek Rossignac for suggesting the name Spirale Reversi. I thank Sir Henry for letting me stay in his beautiful home. I thank UBC Food Services for their yummy cinnamon bun (and only for that).

And I thank everybody who made me stay in Canada.

Martin Isenburg

The University of British Columbia

October 1999

für Omi

Chapter 1

Introduction

Efficiently encoding the topology of triangular meshes has recently been the subject of intense study [16, 25, 24, 26, 21, 3, 10] and many representations have been proposed. The sudden interest in this area is fueled by the emerging demand for transmitting 3D data sets over the Internet (e.g. VRML). Since transmission bandwidth is a scarce resource, compact encodings for 3D models are of great advantage.

This work makes three contributions. First we introduce *Mesh Collapse Compression* (mc-compression), a new algorithm for encoding the topology of triangular meshes. Second we present *Quadrilateral Mesh Collapse Compression* (qmc-compression), a new algorithm for encoding the topology of quadrilateral meshes. The compression scheme shares conceptual ideas with mc-compression, but differs greatly in the details. And third we introduce *Spirale Reversi*, a decoding algorithm that improves on work by Rossignac et al. [21, 22].

Our topology encoding algorithms, mc-compression and qmc-compression, are based on the edge contract operation, which has received attention in the computer graphics community. Hoppe [9, 8] made extensive use of the edge contract operation (unfortunately calling it "edge collapse" [5]) for topology preserving mesh simplification and others followed his approach [2, 6, 20]. But we are not aware of any encoding technique that uses the edge contract operation for efficient mesh topology compression. For compressing meshes that are not composed of triangles little work has been reported.

In the next section we will define what triangle meshes are, as their efficient encoding is the meat of this thesis. And in Section 2 we give a comparative survey of previous work in this area.

1.1 Triangle Meshes

Triangle meshes are commonly used to represent surfaces in computer graphics and computer-aided design and manufacturing (CAD/CAM). In this thesis, a triangle mesh consists of a collection of triangles that must fit together properly: at most two triangles may share a common edge and triangles and edges must have a cyclic order around every vertex.

Triangle meshes can be considered as graphs that have been embedded in a surface. Thus, the neighbourhood of each vertex can be continuously mapped to a plane or to a half-plane. In the language of topology, a triangle mesh is embedded in an orientable 2-manifold with borders.

Common representations for triangle meshes (for example the wavefront

OBJ file format) use two lists: a list of vertices and a list of triangles. The list of vertices contains coordinates that specify a physical location for each mesh vertex. This is referred to as the geometry of the triangle mesh. The list of triangles contains triplets of indices into the vertex list that specify the three vertices of each triangle. This is referred to as the topology of the triangle mesh. We are less concerned with the geometry of a triangle mesh than with its topology.

Notice that a mesh representation such as the above has two drawbacks:

- For triangle meshes with v vertices, the triangle list uses at least $3 \log_2 v$ bits for each triangle. Euler's relation implies that there are approximately twice as many triangles as vertices, giving a total of $6 \log_2 v$ bits per vertex. We will see that a constant number of bits per vertex is possible.

- It is difficult to determine the neighbourhood of a triangle. Adjacency information such as the ordering of triangles and edges around each vertex must be reconstructed, which requires sorting [19]. This information is needed my many applications, so it is best if a mesh representation provides the adjacency relations among the mesh triangles.

Tutte's [28] enumeration of topological triangulations implies that at least 3.24 bits per vertex are needed to be able to encode all planar triangular meshes.

The next section surveys a variety of mesh representations that have been proposed to deal with the above drawbacks.

1.2 Previous work

All efficient compression schemes that have been recently proposed for encoding triangle mesh connectivity [27, 16, 25, 26, 7, 21, 3, 10] follow the same pattern. They encode the mesh through a compact and often interwoven representation of the vertex spanning tree *and* the triangle spanning tree. Neither the triangle nor the vertex tree are by themself sufficient to capture the topology (Rossignac gives a nice example [21]). Usually the schemes start at an arbitrary edge and traverse both the vertices and the triangles of the mesh using a deterministic search strategy (e.g. such as breadth or depth first search). Vertices are encountered along the same spiraling vertex spanning tree by the majority of these schemes [25, 26, 7, 21, 10]. Mesh Collapse Compression follows this pattern.

A different approach to topology encoding was presented by Snoeyink and van Kreveld for Delaunay triangulations [23]. Their scheme uses results by Kirkpatrick [18] and encodes all topology information through a permutation of the vertices. The reconstruction algorithm receives batches of vertices and decodes the triangulations in linear time. Denny and Sohler's work [4] extended this scheme to arbitrary *planar* triangulations. Although the cost of storing the topology is zero, the unstructured order in which the verticed are received and the absence of adjacency information during their decompression prohibits predictive geometry encoding. This makes these scheme overall more expensive.

In the following we will briefly describe all of the compression schemes referenced above that are based on spanning trees. However, we limit this description to the simple mesh case. For the details on how these schemes encode meshes with boundary, with holes, or with handles, we refer the reader to the original reference.

Turan [27] was one of the first to observe that the fact that planar graphs could be decomposed into two spanning trees implied that they could be encoded in a constant number of bits per vertex. He gave an encoding that used 12 bits per vertex.

Keeler and Westbrook improved Turan's scheme for encoding planar graphs. They specialize their encoding of planar graphs and maps [16] to achieve a guaranteed 4.6 bits per vertex (bpv) encoding for simple triangle meshes. They build a triangle-spanning tree by traversing all mesh edges of the dual graph in a counter-clockwise depth-first order starting from an arbitrary initial edge. At its leaves they append all edges of that are not part of the spanning tree. A pre-order listing of all non-leaf nodes that describes the type of their first and second child (only five combinations can occur) is sufficient to reconstruct this tree and its corresponding triangulation.

Taubin and Rossignac have the only scheme that explicitly encodes the vertex spanning tree and the triangle spanning tree of a mesh. Their Topological Surgery method [25] cuts a mesh along a set of edges that corresponds to a spanning tree of vertices. This produces a simple mesh without internal vertices that can be represented by a triangle spanning tree. A rather complicated decoding algorithm can reconstruct the mesh from these two trees. Run-length encoding both trees results in practice in bitrates of around 4 bpv. Rossignac proposed a variaton that increases the observed bitrate, but guarantees an upper bound of 6 bpv.

Touma and Gotsman's Triangle Mesh Compression [26] encodes the degree of each vertex along a spiraling vertex tree with an "add <degree>" code. For each branch in the tree they need an additional "split <offset>" code that specifies the start and the length of the branch. This technique implicitely encodes the triangle spanning tree. They compress the resulting sequence of "add" and "split" commands using a combination of run-length and entropy encoding. This achieves bitrates as low as 0.2 bvp for very regular meshes and around 2 to 3 bpv otherwise. However, the unpredictable offset value of the split commands can lead to non-linear complexity in both encoding/decoding time and bitrate.

Gumhold and Strasser [7] introduce a compressed representation for triangle meshes that is closely related to the Edgebreaker method [21]. Starting with the three edges of an arbitrary triangle as what they call the initial "cut-border", they traverse the triangles of the mesh and include them into this cut-border using four different operations (note: the paper talks about six operations, but for simple meshes only four of them are used). One of these four operations splits the cut-border in two pieces, which is why they called it "split cut-border <offset>". This operation corresponds to the "split <offset>" command of the Touma and Gotsman scheme [26]. The required offset value leads to the same non-linear behaviour, since $\log_2 v$ bits are required to encode it. The other three operations are called “new vertex”, “connect forward”, and “connect backward” and distinguish the three different ways to include a triangle into the cut-border. The reported compression rates vary from 3.5 to 5 bpv, but no upper bound is given.

Rossignac’s Edgebreaker [21] is a simpler and more elegant version of Gumhold and Strasser’s scheme [7] that was independently developed. Since the original paper many improvements have been reported [17, 22, 14], with the most recent being presented in Chapter 4. The compression scheme uses the five operations C, R, L, S, and E to include triangle after triangle into an active boundary, which is intially defined around an arbitrary triangle. The two operations S and E replace the “split cut-border <offset>” operation of Gumhold and Strasser’s scheme, thereby eliminating the need for explicitly encoding the offset value. Instead the decoding algorithm computes all offset values in a preprocessing step. The operations C, R, and L are identical to the “new vertex”, “connect forward”, and “connect backward” operations of Gumhold and Strasser [7].

Improving on the original Edgebreaker decoding scheme [21], which has non-linear time complexity, Rossignac and Szymczak introduced the Wrap&zip decoding [22] that decodes simple meshes in provably linear time. However, for meshes with handles the Wrap&zip scheme needs to perform multiple traversals of all mesh triangles. In Chapter 4 we give an in-depth review of Edgebreaker and Wrap&zip and present the Spirale Reversi decoding scheme, which decodes Edgebreaker encoded meshes in a single pass.

The work by King and Rossignac [17] provides a guaranteed 3.67 bpv encoding for the Edgebreaker scheme. This is currently the lowest worst case bound and lies within 13% of the theoretical lower limit by Tutte [28].

De Floriani et al. [3] presented a scheme similar to Rossignac's and Gumhold and Strasser's work. It avoids the "split cut-border" or the S and E operations altogether by using a "SKIP" command that moves the focus to the next triangle whenever the inclusion of the current triangle would mean a split of the active boundary. Their "VERTEX", "RIGHT", and "LEFT" operation correspond to the C, R, and L operation of Edgebreaker [21] or the "new vertex", "connect forward", and "connect backward" operations of Gumhold and Strasser's scheme [7].

This algorithm works only for extendably shellable triangle meshes [1], which includes all simple meshes. For those a bitrate of 6 bpv is guaranteed and experimental bitrates of 4.1 to 4.5 bpv are reported. Triangle meshes with holes and handles are compressed by partitioning them into shellable patches. This leaves us without upper bound and increases the observed bitrate to 5 bpv and higher. Furthermore it requires the replication of all vertices shared by more than one patch (up to 30%), which is expensive and highly undesireable.

We can classify the Touma and Gotsman scheme [26] as vertex based, Gumhold and Strasser [7], Edgebreaker [21], De Floriani et al. [3] as triangle based and Topologycal Surgery [25] as vertex *and* triangle based. Isenburg will soon present Triangle Fixer [10], a truly edge-based algorithm.

Triangle Fixer has a 6 bpv guaranteed and a 3.9 to 4.2 bpv expected encoding. It has relatively simple extensions towards triangle strip compression [11] and polygon mesh compression [15], which make it interesting.

In the next chapter we introduce Mesh Collapse Compression, a new compression scheme that falls into the vertex-based category. This method is most closely related to that of Touma and Gotsman [26]. We also record a degree for each vertex along a spiraling vertex tree. However, this degree is not necessarily the original degree of the vertex, but rather the degree of the vertex in the moment it is encountered. The algorithm performs a sequence of edge contract operations in the course of the encoding process, which modifies the degree of nearby vertices. Therefore our code words have a slightly higher spread, which affects the efficiency of subsequent entropy encoding. The advantage of Mesh Collapse Compression over Touma and Gotsman's scheme is that we do not have to deal with unpredictably large offset values. Instead of the "split <offset>" code we use a start symbol S and an end symbol E to encode branches in the vertex spanning tree. Applying simple entropy encoding (e.g. Huffman encoding) to our code sequences results in a bitrate of 1 to 4 bpv. A combination of run-length and entropy encoding as it was done by Touma and Gotsman promises even higher compression.

Chapter 2

Mesh Collapse Compression

In this chapter we present a new algorithm for encoding the topology of triangular meshes. Our scheme performs a sequence of edge contract and edge divide operations that collapse the entire mesh into a single vertex. With each edge contraction we store a vertex degree and with each edge division we store a start and an end symbol. This uniquely determines all inverse operations. For meshes that are homeomorphic to a sphere, the algorithm is especially simple. However, the algorithm also encodes surfaces of higher genus at the expense of a few extra bits per handle. A video demonstrating an earlier version of Mesh Collapse Compression can be found in [13].

In the first section we introduce the Mesh Collapse Compression algorithm and prove its correctness. In Section 2.2 we present the results of mc-compressing various example meshes. Some restrictions on the mesh topology that were imposed for the sake of simplicity are lifted in Section 2.3 and in Section 2.4 we summarize our contributions.

2.1 Mesh Collapse Compression

Before we describe the compression scheme, we want to define what properties we expect the input mesh to have:

1. The mesh is a surface composed of topological triangles (e.g. every face is bound by three edges).

2. The mesh has no boundary and no holes (e.g. every edge is bound by two faces).

3. The mesh has no handles (e.g. the mesh is topologically equivalent to a sphere).

Later we will explain how to mc-compress meshes that have a boundary, have holes, or have handles.

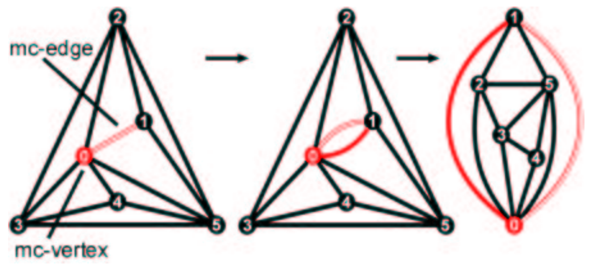


Figure 2.1: Cutting and opening the mc-edge turns the mesh into a digon.

Given a mesh with these properties, the compression scheme initially declares an arbitrary vertex to be the *mc-vertex* and an arbitrary directed edge leaving the mc-vertex to be the *mc-edge*. Then the mesh is cut and

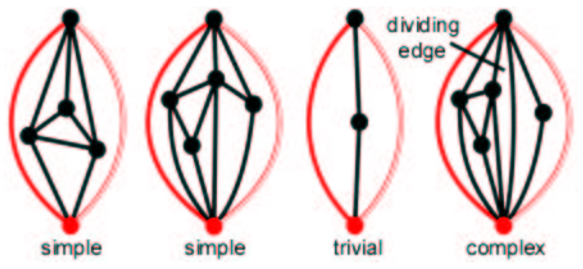


Figure 2.2: Two simple, one trivial, and one complex digon.

opened along the mc-edge, which creates a new face that is bounded by only two edges. For easier illustration we arrange this face to be the outer face as shown in Figure 2.1. The resulting configuration is called a *digon*. This is a triangulation with the exception of the outer face, which is bounded by only two edges.

We distinguish between *trivial* digons, *simple* digons, and *complex* digons: A digon is *trivial* when it has only three vertices. A digon is *simple* when only the two bounding edges connect the two vertices of the outer face. A digon is *complex* when there are more than two edges. Each additional edge is a *dividing edge*. A complex digon with d dividing edges can be divided into $d+1$ simple digons along its dividing edges. This is illustrated in Figure 2.2.

Subsequently the mc-compression algorithm performs a sequence of edge contract and edge divide operations that decomposes the initial digon into one or more trivial digons. We call these two operations *mc-contract* and *mc-divide*.

2.1.1 The mc-contract operation

The mc-contract operation takes a simple digon as input and returns a vertex, a vertex degree, and a digon with one fewer vertex, three fewer edges, and two fewer faces. The resulting digon can be either simple or complex. This operation first contracts the current mc-edge, then removes the resulting loop, and finally selects the next edge counterclockwise around the mc-vertex to be the new mc-edge as illustrated in Figure 2.3.

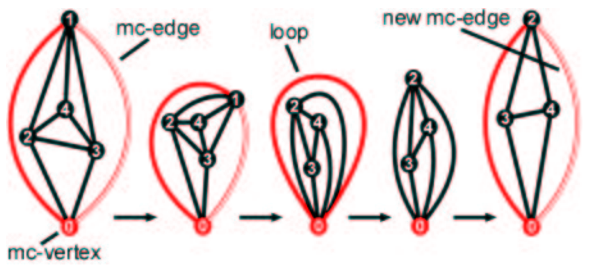


Figure 2.3: An illustration of the mc-contract operation.

The inverse operation is uniquely defined by the removed vertex (e.g. the vertex that collapses into the mc-vertex) and its degree. Contracting the mc-edge moves the edges connected to this vertex over to the mc-vertex. The inverse operation will have to move these edges back. Because the order of the edges is preserved, only their number is important.

The minimal number of edges connected to a vertex is three. The maximal number is theoretically limited only by the total number n of mesh vertices (e.g. degenerated pyramid-shaped meshes can result in a vertex degree as high as $n - 1$). In practice, however, vertex degrees are spread around six.

2.1.2 The mc-divide operation

The mc-divide operation takes a complex digon with d dividing edges as input and returns two digons that have together $d-1$ dividing edges. One of the two resulting digons will always be simple. The other digon will usually be simple too, since complex digons have generally only one dividing edge ($d=1$). However, in case the complex input digon had more than one dividing edge ($d>1$), then one of the output digon will be complex too, but with one fewer dividing edge. In Figure 2.4 is an illustration of the mc-divide operation.

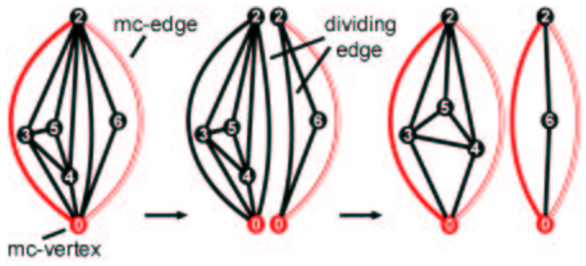


Figure 2.4: An illustration of the mc-divide operation.

2.1.3 Encoding

Starting with the initial digon, an empty digon stack, an empty code stack, and an empty vertex stack we first push the mc-vertex on the vertex stack. Then we repeatedly apply the mc-contract operation until either a complex or a trivial digon is encountered. For each mc-contract operation we push the removed vertex on the vertex stack and its degree on the code stack. When we encounter a complex digon we apply the mc-divide operation. We push a start symbol S on the code stack, push one of the resulting digons on the digon stack and continue the compression process on the other. When we encounter a trivial digon we push two of its three vertices (e.g. not the mc-vertex) on the vertex stack and an end symbol E on the code stack. If the digon stack is empty we terminate. Otherwise we pop a digon from this stack and continue. The recorded information is sufficient to invert each operation. Here is this algorithm in java-like pseudo-code:

```
Codec mc_encode(Mesh mesh) {
    Codec codec = new Codec();
    Digon digon = digonify(mesh);
    codec.pushDigon(digon);
    codec.pushVertex(digon.v0);
    while (codec.hasMoreDigons()) {
        digon = codec.popDigon();
        while (not digon.trivial()) {
            if (digon.complex()) {
                Digon subdigon = mc_divide(digon);
                codec.pushDigon(subdigon);
                codec.pushCode('S');
            }
            else {
                Vertex vertex = mc_contract(digon);
                codec.pushVertex(vertex);
                codec.pushCode(vertex.degree);
            }
        }
        codec.pushVertex(digon.v1);
        codec.pushVertex(digon.v2);
        codec.pushCode('E');
    }
    return codec;
}
```

Note: The vertex that sits at the top of a digon is duplicated by an mc-divide operation. Thus, it seems to be pushed multiple times onto the vertex stack.

However, the actual implementation of the encoding algorithm avoids this by using a simple convention: The vertex that sits at the top of the resulting digon that is processed first is treated as usual (e.g. the next mc-contract operation will pushed onto the vertex step). The duplicate vertex that sits at the top of the other digon is marked and will not be pushed onto the vertex stack. During decoding this situation is detected and dealt with based on the code words in the code stack.

2.1.4 The mc-expand operation

The mc-expand operation is the inverse of the mc-contract operation. It takes a digon, a vertex and a vertex degree as input and returns a simple digon with one more vertex, three more edges, and two more faces. It connects the new vertex twice to the mc-vertex and moves the mc-edge and the next degree -3 edges in counterclockwise order around the mc-vertex over to the new vertex. The last edge is duplicated as illustrated in Figure 2.5. Finally the operation updates the mc-edge.

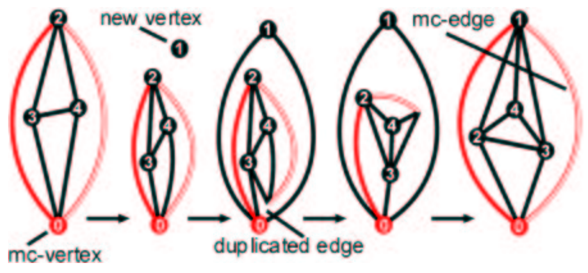


Figure 2.5: An illustration of the mc-expand operation.

2.1.5 The mc-join operation

The mc-join operation is the inverse of the mc-divide operation. It takes two digons as input and returns a complex digon. Usually both input digons are simple and the output digon has one dividing edge. In case the two input digons have already d dividing edges, the output digon will have $d+1$ dividing edges. For an illustration of the mc-join operation read Figure 2.4 from right to left.

2.1.6 Mesh Collapse Trees

Mesh collapse compression performs a sequence of edge contract and edge divide operations that collapses the entire mesh into a single vertex. This implicitly creates a tree with weighted edges. The weights are vertex degrees and capture the topology of the unlabeled mesh. The nodes are vertices and capture the labeling of the mesh. We call this weighted-edge tree an *mc-tree*. Any encoding of the mc-tree constitutes an encoding for the corresponding mesh.

The structure of an mc-tree is reflected in the permutation of vertices and code words in the respective stacks. The start and end symbols S and E on the code stack capture its branching structure, the permutation of vertex degrees on the code stack capture the edge weights along each branch, and the permutation of vertices on the vertex stack capture the node assignment. A complete example is shown in Figure 2.6 using digons (left) and using triangulations (right).

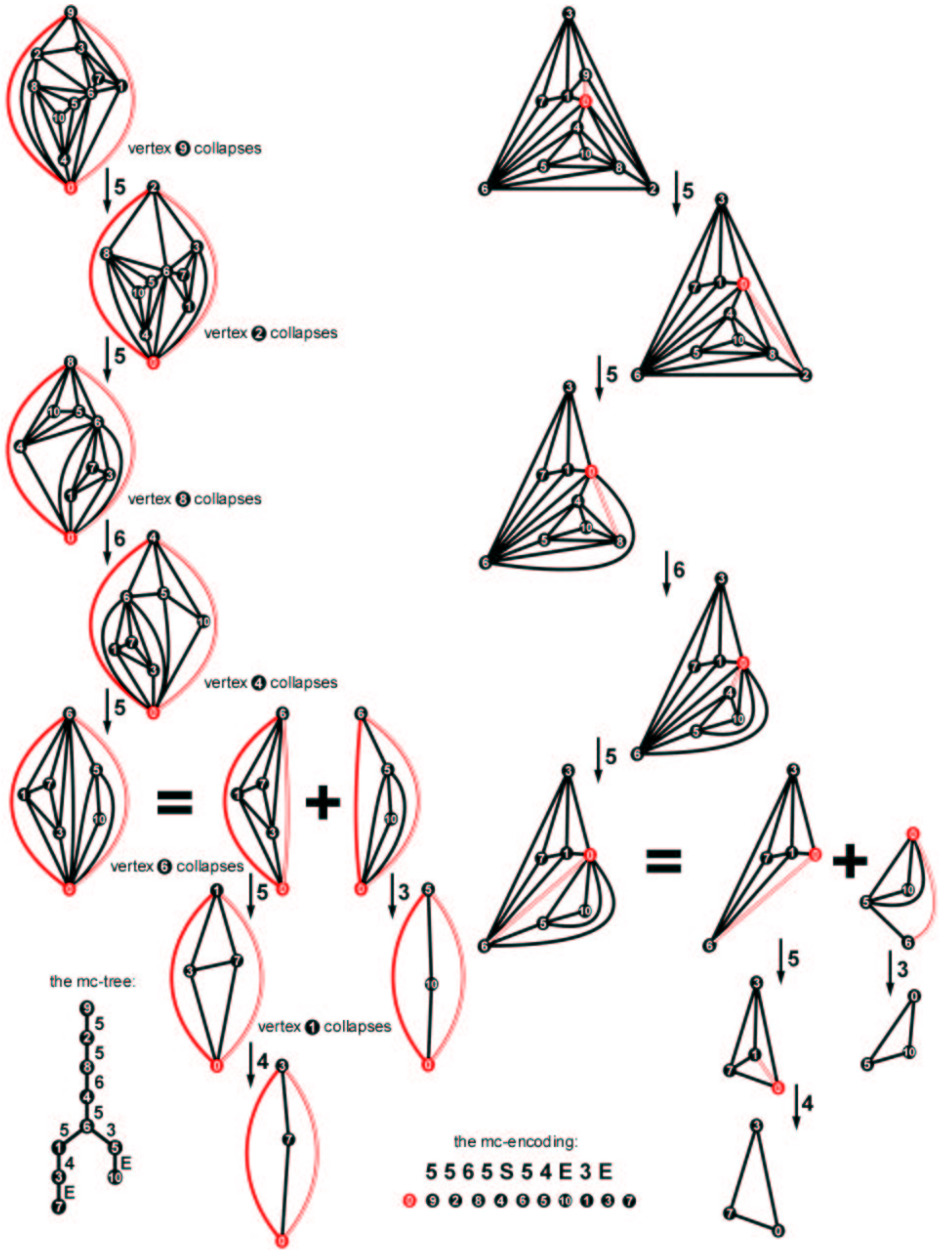


Figure 2.6: A small mesh is mc-compressed with seven mc-contract and one mc-divide operations.

The set of contracted edges is a spanning tree of the vertices and so is the mc-tree if we add the mc-vertex at the root as illustrated in Figure 2.7.

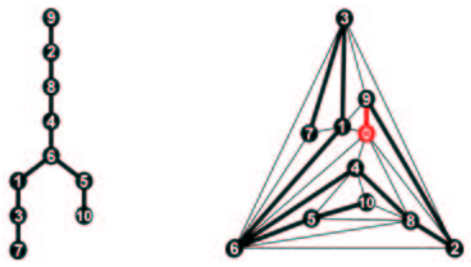


Figure 2.7: The mc-tree and its embedding in the mesh.

2.1.7 Decoding

Starting with an empty digon stack, a non-empty code stack, and a non-empty vertex stack we process the code words in reverse order by popping them from the code stack. If the code word is an end symbol E we push the current digon on the digon stack and create a new trivial digon with the next two vertices from the vertex stack (the mc-vertex is not assigned yet). If the code word is a start symbol S we pop a digon from the digon stack and join it with the current digon using the mc-join operation. Otherwise the code word is a vertex degree and we perform an mc-expand operation to insert the next vertex from the vertex stack into the current digon. We repeat this until all code words are processed. Finally we assign the last vertex left of the vertex stack as the mc-vertex and convert the digon to a mesh. Here is this algorithm in java-like pseudo-code:

```
Mesh mc_decode(Codec codec) {
    Digon digon = null;
    while (codec.hasMoreCodes()) {
        int code = codec.popCode();
        if (code == 'E') {
            codec.pushDigon(digon);
            Vertex v1 = codec.popVertex();
            Vertex v2 = codec.popVertex();
            digon = new Digon(null, v1, v2);
        }
        else if (code == 'S') {
            Digon subdigon = codec.popDigon();
            mc_join(digon, subdigon);
        }
        else {
            Vertex v = codec.popVertex();
            mc_expand(digon, v, code);
        }
    }
    digon.v0 = codec.popVertex();
    return undigonify(digon);
}
```

2.1.8 Proving correctness

In this section we prove that Mesh Collapse Compression encodes a digon of v vertices with exactly $v-3$ mc-operations and that each operation is invertible.

Let us quickly recall the definitions. We start with a *digon* of v vertices. This is a triangulation with the exception of the outer face, which is bounded by only two edges. A digon is *trivial* when it has only three vertices. A digon is *simple* when only the two bounding edges join the two vertices of the outer face. A digon is *complex* when there are more than two edges. Each additional edge is a *dividing edge* along which a complex digon can be divided into simple digons.Every step of mc-compression deals with a digon.

We now prove by induction that mc-compression for a digon of v vertices terminates after $c(v) = a + b = v - 3$ mc-operations with a being the number of mc-contract and b being the number of mc-divide operations.

Termination case ($v = 3$)**:** The digon is trivial. The three vertices are pushed on the vertex stack. The digon can be reconstructed from the order of its vertices on the stack.

Iteration case ($v > 3$)**:** There are two cases depending on whether a digon is simple or complex.

In case A the digon is simple. This digon of v vertices is input to an mc-contract operation, which outputs a digon of $v-1$ vertices. The corresponding vertex degree is pushed on the code stack. The corresponding vertex is pushed on the vertex stack. The digon of v vertices can be reconstructed from the digon of $v-1$ vertices using the vertex degree and the vertex from the respective stacks. The mc-compression process continues with a digon of $v - 1$ vertices.

In case B the digon is complex. This digon of v vertices is input to an mc-divide operation, which outputs two digons of together $v_1 + v_2 = v + 2$ vertices with $v_1 \geq 3$ and $v_2 \geq 3$. The digon of v vertices can be reconstructed from the two digons of v_1 and v_2 vertices. One mc-compression process continues on the digon with v_1 vertices. Another mc-compressions process continues on the digon with v_2 vertices. Markers for seperating the code words produced of the two processes are pushed on the code stack.

Analysis: The axioms that define the number $c(v)$ of mc-operations necessary to mc-compress a digon of v vertices are easily derived from the three cases above:

1. $c(3) = 0$
2. $c(v) = 1 + c(v-1)$
3. $c(v) = 1 + c(w) + c(v-w+2) \quad 3 \leq w \leq v-1$

Using these axioms we now prove by induction that $c(v) = v - 3$. For axiom 1 this is trivial. For axiom 2 and for axiom 3 we use the substitution rule:

Induction Base:	$c(3) = 0$	
Induction Assumption:	$c(k) = k - 3$	for $3 \leq k < v$
Proof with axiom 2:	$c(v) = 1 + c(v-1)$	for $v > 3$
	$= 1 + (v-1) - 3$	
	$= v - 3$	q.e.d.
Proof with axiom 3:	$c(v) = 1 + c(w) + c(v-w+2)$	for $v > 3$
	$= 1 + w - 3 + v - w + 2 - 3$	
	$= v - 3$	q.e.d.

Axiom 2 counts the number a of mc-contract operations and axiom 3 counts the number b of mc-divide operations operations. Hence, the total count $c(v) = v - 3$ is the sum $a + b$ of the two.

During mc-compression a sequence of code words and a sequence of vertices are pushed onto a stack that (a) make every operations invertible and

(b) specify the order in which the operations occured. This constitutes an encoding of the topology of the original digon.

2.2 Results

The results of mc-compressing various example meshes are summarized in Table 2.1. We have two entries for each mesh which were obtained by picking arbitrary initial mc-edges. The code word histograms suggest that we can easily achieve bit-rates of 1 to 4 bits per vertex using a simple entropy encoding (e.g. Huffman encoding). A combination of entropy and run-length encoding as it was done in [26] for similar code sequences promises even more compact encodings.

mesh characteristics		code word histogram											bits p.
name	vrtx/trngl	S	E	3	4	5	6	7	8	9	10	11	vertex
bishop	250/496	1	1	7	35	158	46	0	0	0	0	0	1.572
bishop	250/496	0	0	0	36	182	29	0	0	0	0	0	1.248
bunny	1524/3044	20	20	183	357	446	327	143	34	11	0	0	2.743
bunny	1524/3044	33	33	158	375	448	319	139	41	8	0	0	2.813
shape	2562/5120	0	0	14	147	2228	169	1	0	0	0	0	1.197
shape	2562/5120	0	0	19	143	2221	175	1	0	0	0	0	1.203
triceratops	2832/5660	50	50	248	682	923	657	218	32	12	6	1	2.621
triceratops	2832/5660	57	57	228	708	905	664	216	35	10	6	0	2.638

Table 2.1: Example results of mc-compressing various triangle meshes of sphere topology.

2.3 Boundaries, Holes, and Handles

In this section we lift the restrictions on the mesh topology that were imposed earlier. We allow the input mesh to have a boundary, holes, and/or handles.

2.3.1 Meshes with a boundary or holes

Triangle meshes that have a single boundary or multiple holes are subject to a simple preprocessing step. This preprocessing modifies the mesh and turns it into a triangulation.

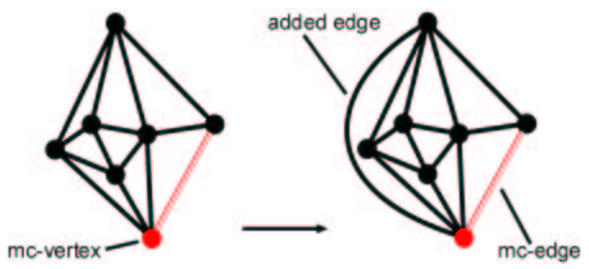


Figure 2.8: Patching a boundary with an additional edge.

A single boundary is patched with additional edges that connect the mc-vertex to other boundary vertices. The mc-edge has to be one of the boundary edges as depicted in Figure 2.8. The number of additional edges is recorded so that they can be removed after decoding in a corresponding postprocessing step.

Multiple holes are patched with one dummy vertex per hole which connects to all vertices around this hole as illustrated in Figure 2.8. These dummy vertices are marked and can be removed in a corresponding postprocessing step.

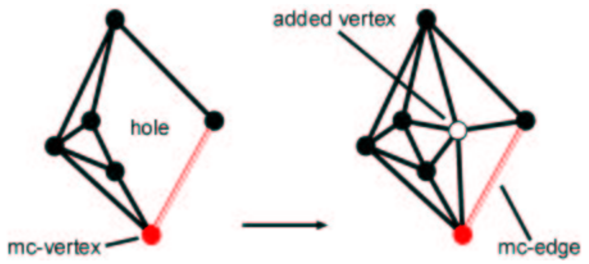


Figure 2.9: Patching a hole with a dummy vertex.

2.3.2 Meshes with handles

The presence of handles in a mesh requires some extra attention. The same algorithm as before is used. But whenever a complex digon is encountered additional cases are possible. By dividing edges separated components of a complex digon can still be connected along a handle. We say such a digon is *connected complex* and is divided into *connected* digons. Dividing a connected complex digon breaks the handle. This configuration is detected, as processing one of the connected digons works its way along the handle and also encodes the other.

Unlike a complex digon, a connected complex digon does not cause a branch but a loop in the mc-tree. This loop is closed when the mc-edge of the other connected digon is encountered. This encounter can happen during an mc-contract operation, during an mc-divide operation, or inside a trivial digon. In either case we record an M symbol followed by two small integers.

The first integer specifies the position of the mc-edge (or rather its corresponding digon) in the stack. This is necessary since multiple handles may not be closed in the order they were broken. Then the corresponding digon is removed from the stack.

The second integer specifies the mc-edge among the edges under consideration. After an mc-contract operation these are the six directed edges (e.g. three undirected edges) that have been removed. After an mc-divide operation these are the two directed edges (e.g. one undirected edge) that are candidates to be pushed onto the stack. Inside a trivial digon these are the eight directed edges (e.g. four undirected edges) that span a trivial digon. This is illustrated in Figure 2.10.

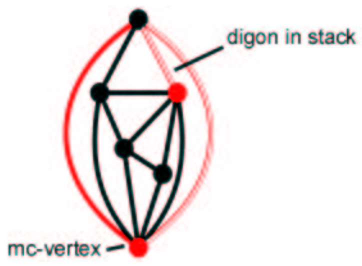


Figure 2.10: Encoding handles in the mesh.

2.4 Summary

We presented a novel encoding scheme for mesh topology. Our algorithm is simpler than approaches by [21, 26, 3] and produces a code sequence similar to [26]. Subsequent run-length and/or entropy encoding results into compact bitstreams of 1 to 4 bits per vertex. This is competitive with the highest compression ratios currently known.

Chapter 3

Quadrilateral Mesh Collapse Compression

In this chapter we present a new algorithm for encoding the topology of quadrilateral meshes. This compression method is related to Mesh Collapse Compression [12], the topology encoding scheme for triangular meshes we introduced in the previous chapter. Little work has been reported on the compression of meshes that are not entirely composed of triangles. Quadrilateral meshes are not as commonly used as triangular meshes, but play an important role in finite element theory and engineering applications.

The algorithm performs a sequence of operations that collapse the entire mesh into a single vertex. With each operation we store a small number that uniquely determines the inverse operation. For meshes that are homeomorphic to a sphere, the algorithm is especially simple. However, the algorithm also encodes surfaces of higher genus at the expense of a few extra bits per handle.

In the first section we introduce the Quadrilateral Mesh Collapse Compression algorithm. In Section 3.2 we present the results of qmc-compressing various example meshes. Some restrictions on the mesh topology that were imposed for the sake of simplicity are lifted in Section 3.3. In Section 3.4 we summarize our contributions.

3.1 Quadrilateral Mesh Collapse Compression

As before we want to define what properties the input mesh is expected to have:

1. The mesh is a surface composed of topological quadrilaterals (e.g. every face is bound by four edges).

2. The mesh has no boundary and no holes (e.g. every edge is bound by two faces).

3. The mesh has no handles (e.g. the mesh is topologically equivalent to a sphere).

Later we will explain how to qmc-compress meshes that have a boundary, holes, or handles. For meshes with a boundary or holes this will be a preprocessing step that modifies the input mesh. For meshes with handles this will be a simple generalization of the qmc-compression scheme.

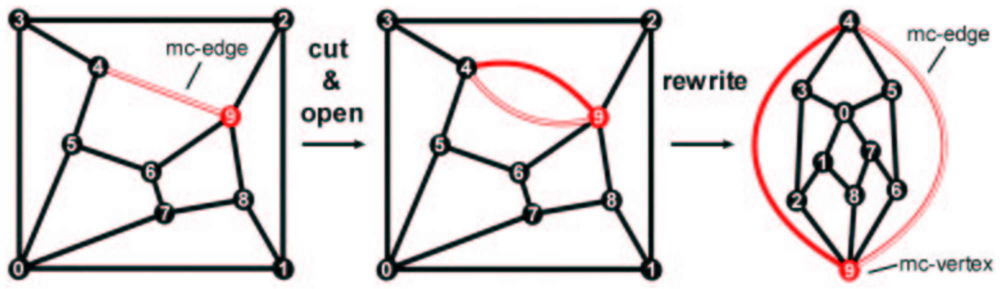


Figure 3.1: Cutting and opening the mc-edge turns the mesh into a diquad.

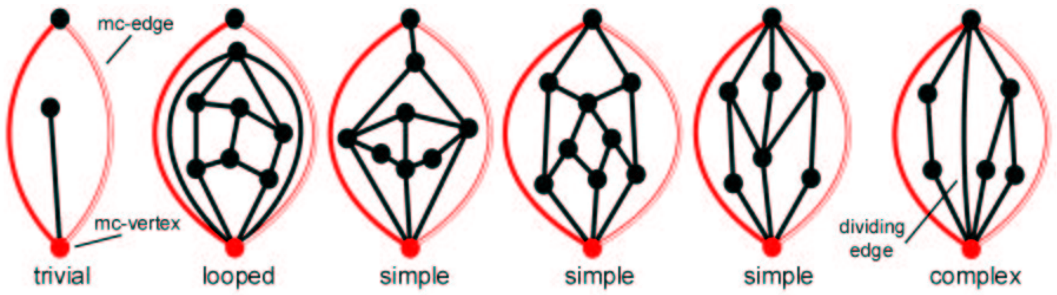


Figure 3.2: One trivial, one looped, three simple and one complex diquad.

3.1.1 Encoding

The algorithm starts off by declaring an arbitrary directed edge to be the *mc-edge*, or *mesh collapse edge* and the vertex it originates from to be the *mc-vertex*, or *mesh collapse vertex.* Every edge in the quadrilateralization adds two possible candidates with opposite direction. Then we cut and open the mesh along the mc-edge. This creates a new face which is bounded by the two copies of the opened mc-edge. Finally we declare the new face to be the outer face and rewrite the mesh respectively. From the two copies of the (directed) mc-edge we declare the one with the outer face on its right to be the new mc-edge. The result is a diquad as illustrated in Figure 3.1. This is a quadrilateralization with exception of the outer face, which is bounded by only two edges.

We distinguish between *trivial* diquads, *looped* diquads, *simple* diquads, and *complex* diquads: A diquad is *trivial* when it has only three vertices. A diquad is *looped* when the destination vertex of the mc-edge has degree two. A diquad is *simple* when it is neither looped, nor complex. A diquad is *complex* when there are more than two edges connecting the two vertices of the outer face. Each additional edge is a *dividing edge.* A complex diquad with d dividing edges can be divided into $d+1$ non-complex diquads. Various diquads are shown in Figure 3.2.

Subsequently the algorithm uses three invertible operations that decompose the initial diquad into one or more trivial diquads:

- The *qmc-deloop* operation takes a looped diquad as input and removes one vertex and one quadrilateral. The resulting diquad can be either trivial, looped, simple, or complex. The operation returns the removed vertex. This information is sufficient to invert the operation.

- The *qmc-contract* operation takes a simple diquad as input and removes one vertex and one quadrilateral. The resulting diquad can be either trivial, looped, simple, or complex. The operation returns the removed vertex and the degree of the removed vertex. Both are needed for inverting this operation.

- The *qmc-divide* operation takes a complex diquad as input and divides it along a dividing edge into two diquads. One of the resulting diquads is guaranteed not to be complex, the other can be either. No information is needed in order to invert the operation.

Starting with the initial diquad the algorithm repeatedly applies the qmc-contract operation on simple digons and the qmc-deloop operation on looped digons until either a trivial or a complex diquad is encountered. For each qmc-contract operation it records the returned vertex and the returned vertex degree. For each qmc-deloop operation it records the returned vertex and the symbol L. When the algorithm encounters a complex diquad the qmc-divide operation is applied. One of the two resulting diquads is pushed onto a stack and the algorithm continues on the other. The symbol S is recorded. When the algorithm encounters a trivial diquad it records two of its vertices (e.g. not the mc-vertex) and the symbol E. If the stack is empty the mc-vertex is recorded and the encoding process terminates. Otherwise the algorithm continues on the diquad popped from the stack.

Here is this algorithm in java-like pseudo-code:

```
Codec qmc_encode(Mesh mesh)
{
    Codec codec = new Codec();
    Diquad diquad = diquadify(mesh);
    codec.pushDiquad(diquad);
    codec.pushVertex(diquad.v0);
    while (codec.hasMoreDiquads()) {
        diquad = codec.popDiquad();
        while (not diquad.trivial()) {
            if (diquad.looped()) {
                Vertex vertex = qmc_deloop(diquad);
                codec.pushVertex(vertex);
                codec.pushCode('L');
            }
            else if (diquad.complex())
            {
                Diquad subdiquad = qmc_divide(diquad);
                stack.pushDiquad(subdiquad);
                codec.pushCode('S');
            }
            else
            {
                Vertex vertex = qmc_contract(diquad);
                codec.pushVertex(vertex);
                codec.pushCode(vertex.degree);
            }
        }
        codec.pushVertex(diquad.v1);
        codec.pushVertex(diquad.v2);
        codec.pushCode('E');
    }
    return codec;
}
```

3.1.2 The qmc-deloop operation

The qmc-deloop operation takes a looped diquad as input and removes one vertex, two edges, and one quadrilateral. The removed vertex has degree two and is the destination vertex of the mc-edge. The two removed edges connect the removed vertex to the mc-vertex. Since one of these edges is the mc-edge, a new mc-edge needs to be selected. This is the next edge counterclockwise around the mc-vertex as illustrated in Figure 3.3a. The resulting diquad is either trivial, looped, simple, or complex. The removed vertex is returned.

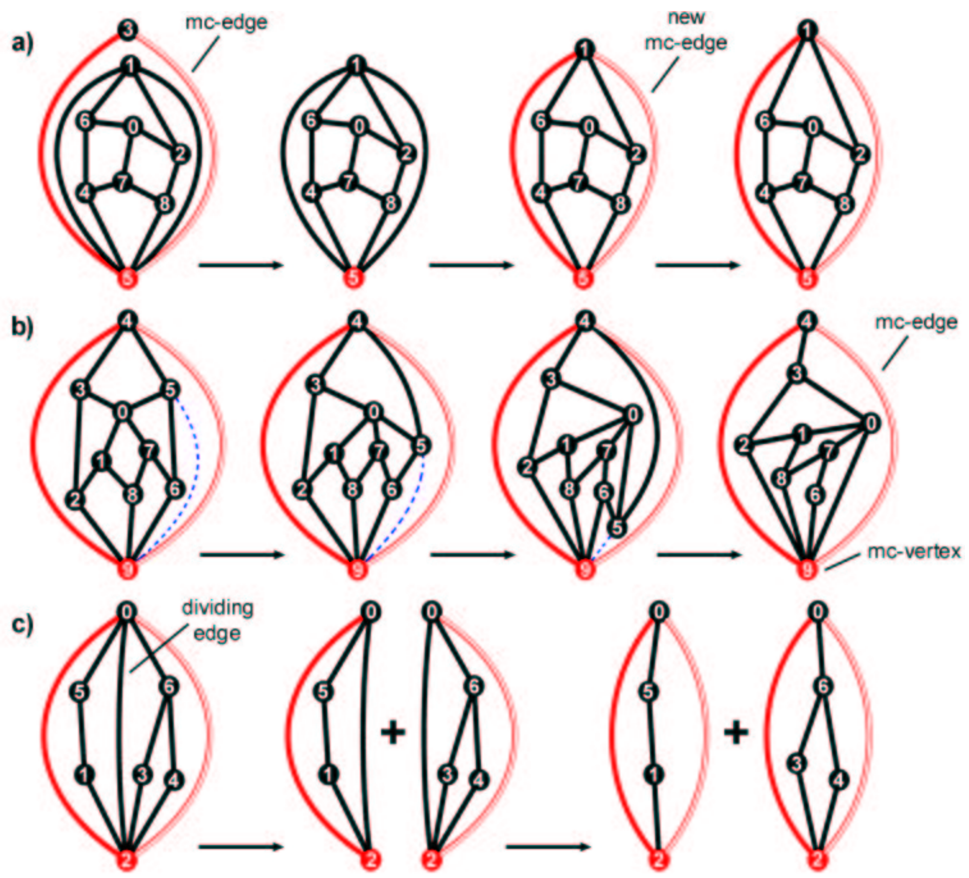


Figure 3.3: An illustration of the qmc-encode operations: (a) the qmc-deloop operation, (b) the qmc-contract operation, and (c) the qmc-divide operation

3.1.3 The qmc-contract operation

The qmc-contract operation takes a simple diquad as input and and removes one vertex, two edges, and one quadrilateral. The removed vertex is at the other end of a diagonal across the quadrilateral left of the mc-edge that leaves the mc-vertex. The two removed edges are both adjacent to the removed vertex. Since the mc-edge is not removed, there is no need to select a new mc-edge. See Figure 3.3b for an illustration. The resulting diquad is either trivial, looped, simple, or complex. The removed vertex and its degree are returned.

3.1.4 The qmc-divide operation

The qmc-divide operation takes a complex diquad with d dividing edges as input and returns one diquad with 0 dividing edges and one with $d-1$ dividing edges. The first will obviously be simple. It is split from the input diquad along a dividing edge. The other will usually be simple too, since the input diquad has generally only one dividing edge ($d = 1$). However, in case the input diquad had more than one dividing edge ($d > 1$), the other diquad will be complex. In Figure 3.3c is an illustration of the qmc-divide operation.

3.1.5 Decoding

We start with an empty stack and process the code words in reverse order. If the code word is an end symbol E we push the current diquad on the stack and create a new trivial diquad with two vertices (the mc-vertex is not assigned yet). If the code word is a start symbol S we pop a diquad from the stack and join it with the current diquad using the qmc-join operation. If the code word is a loop symbol L we apply a qmc-enloop operation to inserts the next vertex into the current diquad. Otherwise the code word is a vertex degree and we perform a qmc-expand operation to insert the next vertex into the current diquad. We repeat this until all code words are processed. Finally we assign the mc-vertex and convert the diquad to a mesh. Here is this algorithm in java-like pseudo-code:

```
Mesh qmc_decode(Codec codec)
{
    Diquad diquad = null;
    while (codec.hasMoreCodes()) {
        int code = codec.popCode();
        if (code == 'E') {
            codec.pushDiquad(diquad);
            Vertex v1 = codec.popVertex();
            Vertex v2 = codec.popVertex();
            diquad = new Diquad(null, v1, v2);
        }
        else if (code == 'S') {
            Digon subdiquad = codec.popDiquad();
            qmc_join(diquad, subdiquad);
        }
        else if (code == 'L') {
            Vertex v = codec.popVertex();
            qmc_enloop(diquad, v);
        }
        else {
            Vertex v = codec.popVertex();
            qmc_expand(diquad, v, code);
        }
    }
    diquad.v0 = codec.popVertex();
    return undiquadify(diquad);
}
```

3.1.6 The qmc-enloop operation

The qmc-enloop operation is the inverse of the qmc-deloop operation. It takes a diquad and a vertex as input and returns a looped diquad with one more vertex, two more edges, and one more quadrilateral. See Figure 3.4a for an illustration.

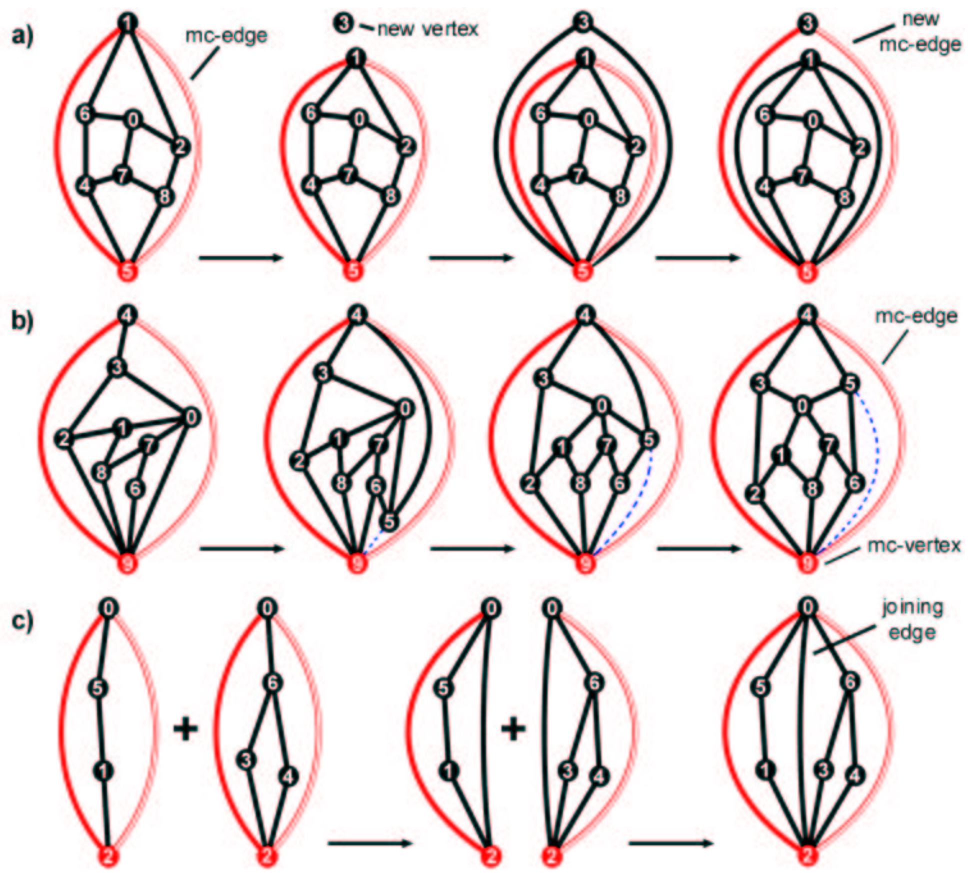


Figure 3.4: An illustration of the qmc-decode operations: (a) the qmc-enloop operation, (b) the qmc-expand operation, and (c) the qmc-join operation

3.1.7 The qmc-expand operation

The qmc-expand operation is the inverse of the qmc-contract operation. It takes a diquad, a vertex and a vertex degree as input and returns a simple diquad with one more vertex, two more edges, and one more quadrilateral. See Figure 3.4b for an illustration.

3.1.8 The qmc-join operation

The qmc-join operation is the inverse of the qmc-divide operation. It that takes two diquads as input and returns a complex diquad. Usually both input diquads are non-complex and the output diquad has one dividing edge. In case the two input diquads have already d dividing edges, the output diquad will have $d + 1$ dividing edges. For an illustration of the qmc-join operation see Figure 3.4c.

3.2 Results

The result of qmc-compressing various example meshes are summarized in Table 3.1. These meshes are randomly generated and approximate the surface of a sphere. The code word histograms suggest that we can easily achieve bit-rates of 1 to 3 bits per vertex using a simple entropy encoding (e.g. Huffman encoding). A combination of entropy and run-length encoding as it was done in [26] for similar code sequences promises even more compact encodings.

3.3 Boundaries, Holes, and Handles

In this section we lift the restrictions on the mesh topology that were imposed earlier. We allow the input mesh to have a boundary, holes, and/or handles.

mesh characteristics		code word histogram												bits p.
name	vrtx/qdrltrl	L	S	E	2	3	4	5	6	7	8	9	10	vertex
davis1	590/588	275	17	17	0	198	12	31	27	14	7	4	2	2.34
davis2	1230/1228	481	41	41	0	466	41	65	60	46	14	9	4	2.61
davis3	3120/3118	1477	77	77	0	1132	46	125	137	83	17	18	5	2.19
davis4	4670/4668	2254	114	114	0	1575	80	201	215	140	56	26	6	2.25
davis5	5990/5988	2834	158	158	0	2009	106	263	296	184	99	29	9	2.32

Table 3.1: Example results of qmc-compressing various quadrilateral meshes of sphere topology.

3.3.1 Meshes with a boundary or holes

Quadrilateral meshes that have a boundary or holes are subject to a simple preprocessing step. This preprocessing modifies the mesh and turns it into a quadrilaterization. There is however one limitation: Only holes with an even number of vertices along the opening are admissable. The compression scheme, as presented here, is not capable of handling holes with an odd number of vertices. More recent work of the authors [15] deals with such meshes.

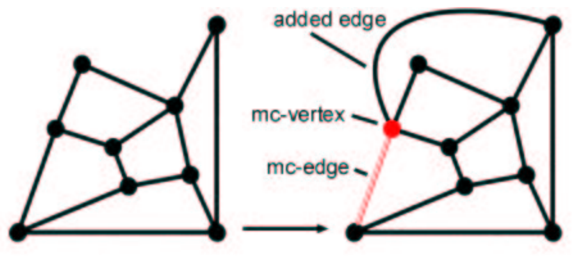


Figure 3.5: Patching a boundary with an additional edge.

For meshes with a boundary we pick an edge on the boundary as the inital mc-edge. Then we add additional edges that connect the mc-vertex to every other boundary vertex in clockwise direction as illustrated in Figure 3.5. Recording the number of these additional edges is sufficient to remove them after decoding in a corresponding postprocessing step.

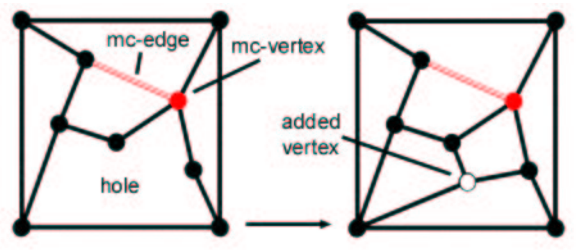


Figure 3.6: Patching a hole with a dummy vertex.

For meshes with holes we use dummy vertices that close each hole. This is illustrated in Figure 3.6. These vertices are marked and can be removed after decoding. Topologically a mesh boundary is just another hole and could also be encoded this way.

3.3.2 Meshes with handles

The presence of handles in a mesh requires some extra attention. The same algorithm as before is used. But whenever a complex diquad is encountered additional cases are possible. By dividing edges seperated components of a complex diquad can still be connected along a handle. We say such a diquad is *connected complex* and is divided into *connected* diquads. Dividing a connected complex diquad breaks the handle. This configuration is detected, as processing one of the connected diquads works its way along the handle and also encodes the other.

Unlike a complex diquad, a connected complex diquad does not cause a branch but a loop in the mc-tree. This loop is closed when the mc-edge of the corresponding other connected diquad is encountered. This encounter can happen during an mc-contract operation, during an mc-divide operation, or inside a trivial diquad. In either case we record an M symbol followed by two small integers.

The first integer specifies the position of the mc-edge (or rather its corresponding diquad) in the stack. This is necessary since multiple handles may not be closed in the order they were broken. Then the corresponding diquad is removed from the stack.

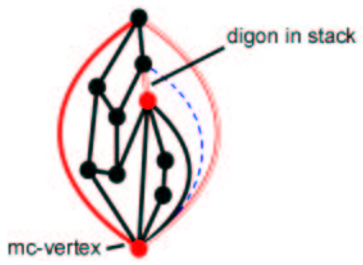


Figure 3.7: Encoding handles in the mesh.

The second integer specifies the mc-edge among the edges under consideration. After a qmc-contract operation these are the six directed edges (e.g. three undirected edges) that have been removed. After a qmc-divide operation these are the two directed edges (e.g. one undirected edge) that are candidates to be pushed onto the stack. Inside a trivial diquad these are the eight directed edges (e.g. four undirected edges) that span this diquad. This is illustrated in Figure 3.7.

3.4 Summary

We presented a novel encoding scheme for encoding the topology of quadrilateral meshes. Our algorithm is simple and produces a sequence of small numbers. Subsequent run-length and/or entropy encoding results into very compact bitstreams of 1 to 3 bits per vertex. This is very competative and rivals with the highest compression ratios currently known.

Chapter 4

Spirale Reversi

In this chapter we present a simple linear time algorithm for decoding Edgebreaker encoded triangle meshes in a single traversal. The Edgebreaker compression technique, introduced in [21], encodes the topology of meshes homeomorphic to a sphere with a guaranteed 2 bits per triangle or less. The encoding algorithm visits every triangle of the mesh in a depth-first order. The original decoding algorithm [21] recreates the triangles in the same order they have been visited by the encoding algorithm and exhibits a worst case time complexity of $O(n^2)$. More recent work [22] uses the same traversal order and improves the worst case to $O(n)$. However, for meshes with handles multiple traversals are needed during both encoding and decoding. We introduce here a simpler decompression technique that performs a single traversal and recreates the triangles in reverse order.

In the next section we briefly summarize the Edgebreaker encoding scheme. A detailed description of the algorithm can be found in [21]. The

Edgebreaker decoding scheme is covered in Section 4.2 and the Wrap&zip decoding scheme is covered in Section 4.3. We introduce our Spirale Reversi decoding scheme in Section 4.4. In these sections we assume that the input mesh has no boundary, no holes, and no handles. Later we explain how encoding and decoding generalizes to meshes with boundary in Section 4.5, with holes in Section 4.6 and with handles in Section 4.7.

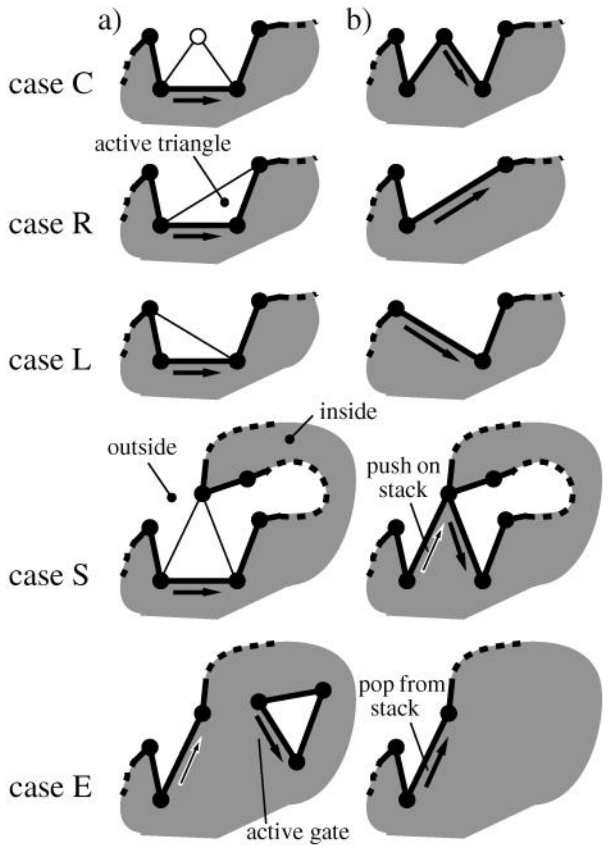


Figure 4.1: The Edgebreaker encoding operations C, L, E, R, and S.

4.1 Edgebreaker encoding

Before we describe the Edgebreaker encoding scheme, we want to define what properties the input mesh is expected to have:

1. The mesh is a surface composed of topological triangles (e.g. every face is bound by three edges).
2. The mesh has no boundary and no holes (e.g. every edge is bound by two faces).
3. The mesh has no handles (e.g. the mesh is topologically equivalent to a sphere).

Later we will describe how the Edgebreaker encoding scheme deals with meshes that have a boundary, have holes, or have handles.

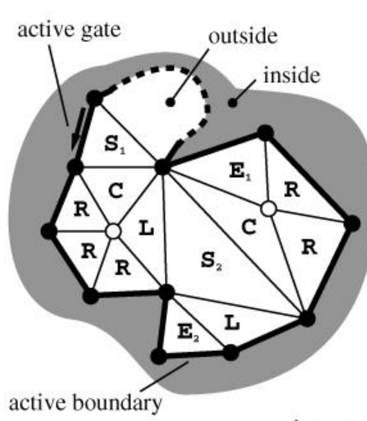
The Edgebreaker encoding process starts with a triangulated mesh and produces a *CLERS string*. It visits every triangle of the mesh by including it into an *active boundary*. Initially the active boundary is an arbitrary triangle of the mesh. The encoding uses five different operations called C, L, E, R, and S to include a triangle into the active boundary. Which operation is chosen depends on how the respective triangle is attached to the active boundary. This expands (operation C), shrinks (operation R and L), splits (operation S), or terminates (operation E) the active boundary. The sequence of C, L, E, R, and S operations describes the traversal of the triangles of the mesh. The corresponding CLERS string is a compact encoding of the topology of the mesh. Now the details:

The encoding process starts off with defining an arbitrary triangle in the mesh to be initial active boundary. The three vertices of the triangle become *boundary vertices* and the three edges of the triangle become *boundary edges*. The boundary edges are directed clockwise around the triangle. The triangle itself is declared to be *inside* of the boundary; the remaining mesh is declared to be *outside* of the boundary. Initially this boundary is the only element in a stack of boundaries. The active boundary is always the top element of this stack.

One of the three initial boundary edges is defined to be the *gate* of the boundary. The gate is directed in the same way as the boundary edges. The adjacent triangle right of the gate is inside, the adjacent triangle left of the gate is outside of the boundary. The *active gate* is the gate of the active boundary. The *active triangle* is the adjacent triangle left of the active gate.

The essential element of the Edgebreaker encoding scheme is: With every operation the active triangle moves from outside to inside of the active boundary. The invariant of the Edgebreaker encoding scheme is: A triangle that lies outside of some boundary is not yet encoded. A triangle that lies inside of all boundaries is already encoded. The Edgebreaker encoding terminates after exactly $t - 1$ operations, with t being the number of triangles of the mesh. Every triangle is processed by one operation with exception of the one that defines the initial active boundary.

The active triangle is included into the active boundary with one of the five operations C, L, E, R, or S. Which operation is chosen depends on how the active triangle is attached to the active boundary. If its third vertex is not on the active boundary then operation C is used. If its third vertex is the next boundary vertex on the active boundary then operation R is used. (Remember that the boundary edges are directed clockwise around the inside.) If its third vertex is the previous boundary vertex on the active boundary then operation L is used. If its third vertex is some other boundary vertex on the active boundary then operation S is used. If its third vertex is the previous *and* the next boundary vertex on the active boundary then operation E is used. This can only happen for an active boundary of length three. See also the illustration in Figure 4.1a.



code	boundary length change	for s_1	for s_2
S_1	+1	*0	
C	+1	1	
R	-1	0	
R	-1	-1	
R	-1	-2	
L	-1	-3	
S_2	+1	-2	*0
L	-1	-3	-1
E_2	-3	-6	-4
C	+1	-5	
R	-1	-6	
R	-1	-7	
E_1	-3	-10	

*start of offset computation

Figure 4.2: Computing the offsets of the S operation for the Edgebreaker decoding.

Each operation requires an update of the active boundary, since the active triangle moves from the outside to the inside of the boundary. The active boundary is expanded (operation C), is shrunk (operation R and L), is split (operation S), or is terminated (operation E). Each operation also requires an update of the active gate, since it moves with the active triangle to the inside of the boundary. Figure 4.1b illustrates the necessary updates. They are as follows:

- The **C operation** inserts one new boundary vertex, inserts two new boundary edges, and removes one boundary edge. The old gate is the removed boundary edge, the new gate is one of the inserted boundary edges. It is on the left as seen from the old gate.

- The **R and L operation** both remove one boundary vertex, remove two boundary edges, and insert one new boundary edge. The new gate is the inserted boundary edge, the old gate is one of the deleted boundary edges. The two operations differ by whether the old gate is on the right (R) or on the left (L) as seen from the new gate.

- The **S operation** splits the active boundary into two boundaries that share one boundary vertex. It inserts two new boundary edges and removes one boundary edge. The total count of boundary vertices increases by one because the shared boundary vertex is counted twice. Both inserted boundary edges become a gate for the respective boundary. The current top element of the boundary stack is popped and the two boundaries are pushed onto the stack. The new top element becomes the active boundary.

- The **E operation** removes the last three boundary vertices and the last three boundary edges. The current top element of the boundary stack is popped. If the stack is empty the encoding ends. Otherwise the new top element becomes the active boundary.

The Edgebreaker encoding scheme as presented so far captures the topology of an unlabeled mesh. Together with the right permutation of the vertex data it captures the topology of a labeled mesh. The vertex data, such as coordinates, texture information, or surface normal, are stored in the order in which the vertices are encountered during the encoding process. Vertices are encountered in the moment they are inserted into the active boundary. For the first three vertices this happens at the start of the Edgebreaker encoding when the initial boundary is defined. For all other vertices this happens during a C operation.

For triangle meshes with v vertices and t triangles that are homeomorphic to a sphere t equals $2v - 4$. The traversal of the mesh triangles reaches new vertices only with the C operation. Since there are twice as many triangles than vertices, half of all operations will be of type C. A straight-forward encoding that encodes a C operation with one bit and the remaining four operations with three bits is guaranteed to use no more than $2t$ or $4v$ bits. A more elaborate encoding of the CLERS sequence guarantees an even lower bound of $3.67v$ bits [17].

The detailed example in Figure 4.11 leads step by step through the final twelve operations of Edgebreaker encoding a mesh.

4.2 Edgebreaker decoding

The Edgebreaker decoding process starts with a CLERS string and produces a triangulated mesh. Two traversals of the CLERS string are needed: A preprocessing phase that computes an offset value for every S operation. A generation phase that creates the triangles in the order in which they were encoded by the Edgebreaker encoding process.

The preprocessing phase computes an offset value for every S operation. The Edgebreaker encoding uses the S operation whenever the third vertex of the active triangle is a vertex on the active boundary other than the previous or the next. In this case, the active boundary is split into two boundaries with this third vertex appearing in both. When the Edgebreaker decoding creates this triangle, it needs to know which vertex on the active boundary to use as the triangle's third vertex. The offset value that is computed during the preprocessing phase is the distance between the active gate and this vertex along the active boundary.

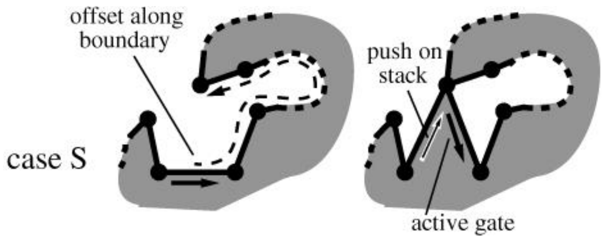


Figure 4.3: Using the offset of the S operation during the Edgebreaker decoding.

The computation of these offset values is very simple. The resulting change in boundary length is added up for all operations following an S operation until and including its corresponding E operation. Since pairs of S and E operations are always nested, the offset values for all S operations can be computed in a single traversal. See also the illustration in Figure 4.2.

The generation phase starts with creating the initial triangle. The active boundary and the gate are identified and the CLERS string is processed. What follows is an almost exact replay of the encoding algorithm. With every operation a new triangle is created and included into the active boundary. The triangle is always attached to the left of the active gate. Which vertex is used as the triangle's third vertex depends on the current operation. Only for the C operation a new vertex is created. For all other operations a vertex from the active boundary is used. For the R operation this is the next and for the L operation this is the previous vertex on the active boundary. For the S operation it is some other boundary vertex. The precomputed offset value specifies its distance from the active gate along the boundary. When the E operation occurs, the active boundary consists of only three boundary vertices. This leaves no choice for the third vertex.

The five operations of the Edgebreaker decoding perform the same updates on boundary and gate as those of the Edgebreaker encoding (see Figure 4.1). Only the S operation is more complex. It uses the precomputed offset to locate the third vertex for the newly created triangle as illustrated in Figure 4.3.

The Edgebreaker decoding scheme as presented so far reconstructs the topology of the unlabeled mesh. Using the vertex permutation that is produced by the Edgebreaker encoding, the mesh labeling is reconstructed. The vertex data is assigned to unlabeled vertices in the order in which they are encountered. Vertices are encountered in the moment they are inserted into the active boundary. For the first three vertices this happens at the start of the Edgebreaker decoding when the initial boundary is defined. For all other vertices this happens during a C operation.

Although in practice only a small fraction of operations are of type S, they imply an asymptotic worst case time complexity of $O(n^2)$ for the Edgebreaker decoding, if the active boundary is maintained in a linear data structure. Each S operation requires a linear search for the vertex specified by the offset. This cost may be reduced to $O(n \log n)$ if the active boundary is maintained in a data structure with a logarithmic instead of a linear search time. However, the more complex update operations of a data structure with logarithmic search time (such as a balanced binary tree) would increase the expected complexity from $O(n)$ to $O(n \log n)$.

The detailed example in Figure 4.12 leads step by step through the final twelve operations of Edgebreaker decoding a mesh.

4.3 Wrap&zip decoding

The Wrap&zip decoding process starts with a CLERS string and produces a triangulated mesh. Only one traversal of the CLERS string is needed. It starts with creating the initial triangle. The active boundary and the gate are identified and the CLERS string is processed. What follows is a modified replay

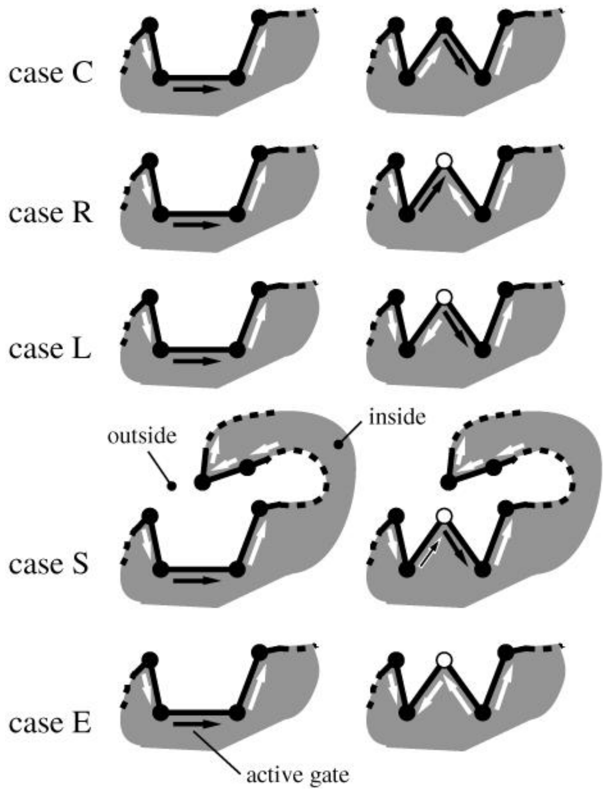


Figure 4.4: The Wrap&zip decoding operations C, L, E, R, and S.

of the encoding algorithm. With every operation a new triangle is created. The triangle is always attached to the left of the active gate. The decision which vertex is the triangle's third vertex is postponed for all operations but the C operation. Instead of some boundary vertex from the active boundary a dummy vertex is used for operations of type L, E, R, and S. This is the wrapping part of the Wrap&zip decoding. For the C operation nothing changes. Like before a newly created vertex is used.

All boundary edges except for the gate have an additional direction assigned that depends on the operation that created them. This *zip direction* is used for the zipping part of the Wrap&zip decoding. Which operation

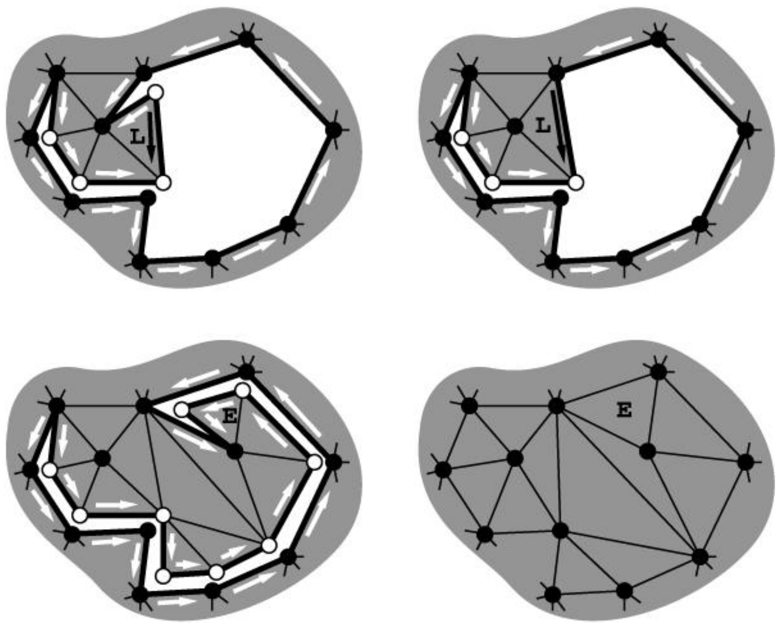


Figure 4.5: Single zipping after an L operation (top) and recursive zipping after an E operation (bottom).

assigns which zip direction is shown in Figure 4.4.

Each time the zip directions of two adjacent boundary edges point to a common vertex, they are zipped together by identifying their other ends. This zipping continues recursively if the resulting vertex exhibits the same property. Whether a zip is necessary needs only to be checked after L and E operations. No immediate zipping is possible after C, R, and S operations. A zip after an L operation never starts recursive zipping, whereas a zip after an E operation always starts recursive zipping. A small example in Figure 4.5 illustrates single zipping after an L and recursive zipping after an E operation.

The Wrap&zip decoding scheme as presented so far reconstructs the topology of the unlabeled mesh. The mesh labeling is reconstructed in the same way as in the Edgebreaker decoding.

The wrapping and zipping technique of this decoding scheme improves on the asymptotic worst case time complexity $O(n^2)$ of the original Edgebreaker decoding. It can be shown that the number of zip operations equals the number of edges in the vertex-spanning tree. Therefore the decoding algorithm has linear time complexity.

The detailed example in Figure 4.13 leads step by step through the final twelve operations of Wrap&zip decoding a mesh.

4.4 Spirale Reversi decoding

The Spirale Reversi decoding process starts with a CLERS string and produces a triangulated mesh. Only one *reverse* traversal of the CLERS string is needed. This completely eliminates the overhead for the S and E operation pairs that is necessary for the Edgebreaker and the Wrap&zip decoding. It can be seen as a step by step reversal of the Edgebreaker encoding.

The Spirale Reversi decoding scheme uses the same boundary definitions as the Edgebreaker encoding scheme. It starts with creating an unlabeled triangle as the initial boundary. It is unlabeled in the sense that no physical vertex is yet associated with the three boundary vertices. The triangle itself is declared to be outside of the boundary. The boundary edges are directed counterclockwise around this triangle.

One of the three boundary edges is defined as the initial active gate. Inside of the boundary is right of the gate, outside of the boundary is left of the gate. The Edgebreaker encoding was growing the inside until there was

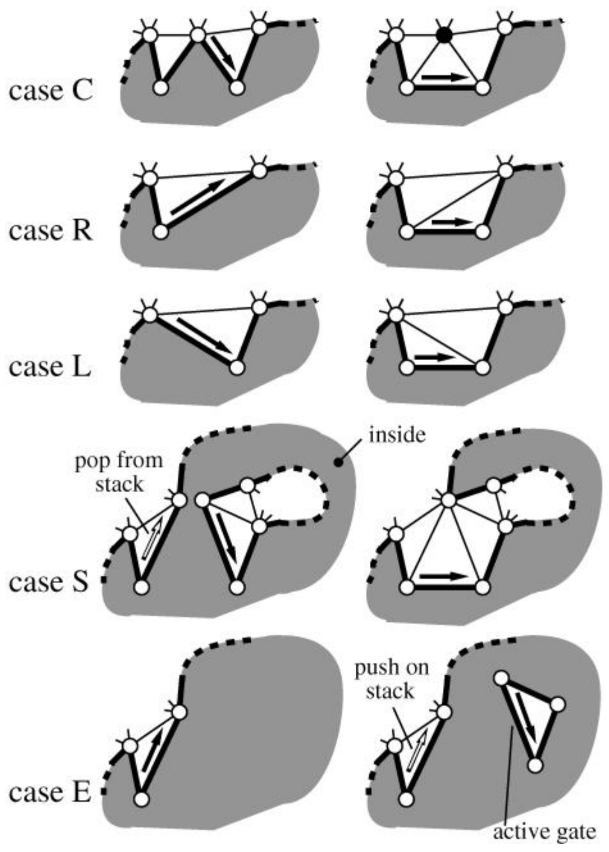


Figure 4.6: The Spirale Reversi decoding operations C, L, E, R, and S.

no triangle left outside. The Spirale Reversi decoding however is growing the outside until there is no triangle left inside. This reflects the *reverseness* of the Spirale Reversi decoding.

The essential element of the Spirale Reversi decoding scheme is: After every operation the triangle left of the active gate has moved from inside to outside of the active boundary. The invariant of the Spirale Reversi decoding scheme is: A triangle that lies outside of some boundary is already decoded. A triangle that lies inside of all boundaries is not yet decoded. The CLERS sequence is processed in the reverse order. Depending on the processed operation the active boundary is shrunk (operation C), is expanded (operation R and L), is merged with the next boundary on the stack (operation S), or is created new (operation E).

Reversing the encoding algorithm works as follows: With every operation a new triangle is created. This triangle is always attached to the right of the active gate. Which vertex is the triangle's third vertex depends on the type of the operation. For the C operation it is the previous boundary vertex on the active boundary. For the R and the L operation a new but unlabeled vertex is created. For the S operation it is a vertex from the boundary that is in the boundary stack directly below the active boundary. More exactly it is the vertex at the origin of this boundary's gate. The vertex at the destination of this boundary's gate and the vertex at the origin of the active gate need to be identified. For the E operation three new unlabeled vertices are created that form a new active boundary in the same way as during initialization.

The required updates of the boundary and of the gate are as follows:

- The **C operation** removes one boundary vertex, removes two boundary edges, and inserts one new boundary edge. The new gate is the inserted boundary edge, the old gate is one of the removed boundary edges. It is on the left as seen from the new gate.

- The **R and L operation** each insert one new boundary vertex, insert two new boundary edges, and remove one boundary edge. The old gate is the removed boundary edge, the new gate is one of the inserted boundary edges. The two operations differ by whether the new gate is on the right

(R) or on the left (L) as seen from the old gate.

- The **S operation** merges the active boundary with the boundary that is directly below in the boundary stack. Thereby one boundary vertex from each boundary are identified into one boundary vertex. It removes two boundary edges and inserts one boundary edge. The total count of boundary vertices decreases by one because the two identified boundary vertices are only one count. Both removed boundary edges are old gates of the respective boundary. The new gate is the inserted boundary edge. The two top elements of the boundary stack are popped and the merged boundaries is pushed onto the stack.

- The **E operation** creates a new active boundary. It inserts three new boundary vertices and three new boundary edges. The new gate is any of the three boundary edges. The new active boundary is pushed on the boundary stack.

The Spirale Reversi decoding scheme as presented so far reconstructs the topology of the unlabeled mesh. Using the reverse of the vertex permutation that is produced by the Edgebreaker encoding, the mesh labeling is reconstructed. The vertex data is assigned to unlabeled vertices in the order in which they are abandoned. Vertices are abandoned in the moment they are removed from the active boundary. For the last three vertices this happens at the end of the Spirale Reversi decoding. For all other vertices this happens during a C operation.

The detailed example in Figure 4.14 leads step by step through the first twelve operations of Spirale Reversi decoding a mesh.

4.5 Handling boundaries

The Edgebreaker approach is capable of encoding the connectivity of any simple triangle mesh without holes. The scheme can easily be made capable of handling a single hole. A triangle mesh with boundary is a triangle mesh with a single hole.

Instead of selecting the loop of edges oriented clockwise around an arbitrary mesh triangle as the initial active boundary, we select the loop of edges oriented clockwise around the hole. Like before, an arbitrary edge from this boundary is declared to be the initial active gate. The vertex data of all boundary vertices is stored in counterclockwise order around the hole starting at the active gate. From there Edgebreaker encoding proceeds like before.

Both the Edgebreaker decoding and the Wrap&zip decoding need additional information to decode the boundary case. They need to know the length of the initial boundary loop (e.g. the length of the hole). This can be precomputed during an initial traversal of the CLERS string. The Spirale Reversi decoding needs no additional information. After decoding the last label of the reversed CLERS string, the active boundary loops around the hole. Then the boundary vertices are simply assigned their data in the opposite order as they were stored during encoding.

4.6 Handling holes

For every additional hole the Edgebreaker encoding runs into a situation in which the third vertex of the active triangle lies on the boundary of a hole. For this scenario the M operation is introduced. The active boundary is merged with the boundary of the hole by opening both at their common vertex and reconnecting them as depicted in Figure 4.7. The vertex data of all boundary vertices is stored in counterclockwise order around the hole starting at the common vertex. In addition to the label M of the operation the following information needs to be recorded:

- A **length** that specifies the number of vertices on the boundary of the hole.

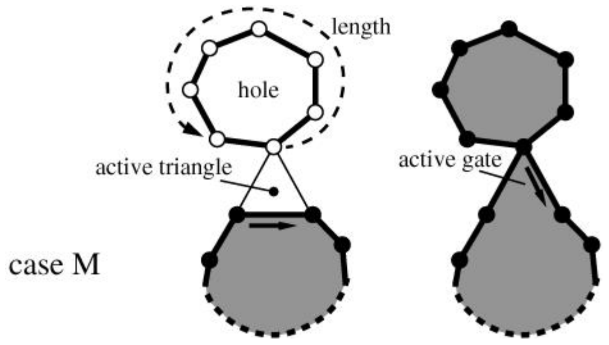


Figure 4.7: The Edgebreaker encoding operation M.

The decoding of a hole is straightforward for all three decoding algorithms. When a label M is processed the associated length value is used to update the boundary accordingly. This involves assigning the data to all vertices around the hole. In Figure 4.7 the Spirale Reversi decoding of a hole is illustrated.

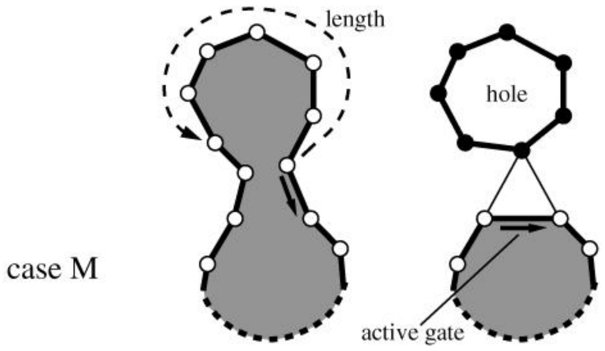


Figure 4.8: The Spirale Reversi decoding operation M.

4.7 Handling handles

For every handle the Edgebreaker encoding runs into a situation in which the third vertex of the active triangle is not on the active boundary, but on some other boundary in the stack. For this scenario the M' operation is introduced. This operation merges these two boundaries into one by opening both at their common vertex and reconnecting them as depicted in Figure 4.9. The respective boundary is then removed from the stack.

In addition to the label M' of the operation three integers are recorded. We modified the original Edgebreaker encoding by the last integer. This will allow to decode a mesh with handles using just a single reverse traversal of the CLERS string. The three integers are as follows:

- An **index** that specifies the respective boundary within the stack of boundaries.
- An **offset1** that specifies the counterclockwise distance between the common vertex and the gate of the boundary from the stack.

- An **offset2** that specifies the counterclockwise distance between the gate of the boundary from the stack and the common vertex.

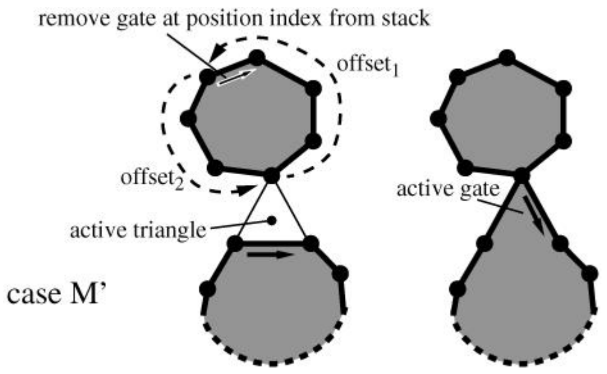


Figure 4.9: The Edgebreaker encoding operation M'.

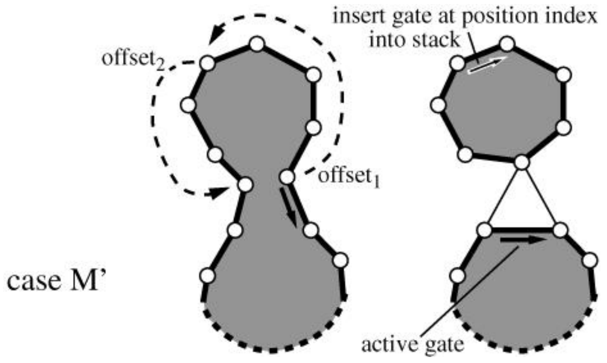


Figure 4.10: The Spirale Reversi decoding operation M'.

The original Edgebreaker decoding uses the three integers it associates with the M' operation to replay the situation encountered during the encoding. The decoding cost per M' operation is $O(n)$. However, the number of M operations is bound by the genus of the mesh and generally small. Neither Wrap&zip nor Spirale Reversi decoding aim at improving the worst case time complexity for the M' operation.

For the Wrap&zip decoding of meshes with handles the authors [22] had to modify the Edgebreaker encoding. The modified approach is more complicated and requires three instead of one traversal of the mesh triangles. For details we refer to the original reference [22].

The Spirale Reversi decoding of the M' operation follows the concept of reversing the encoding process. The two offsets specify the split of the active boundary and the position of the gate in the boundary that is inserted into the stack. The index specifies the position at which this boundary is inserted into the stack. This is illustrated in Figure 4.10

4.8 Summary

We presented a simple linear time algorithm for decoding Edgebreaker encoded triangle meshes. The concept of reversing the encoding process allows to decode a mesh with a single traversal of the CLERS string. For meshes without handles our scheme eliminates the need for the look-ahead procedure used by the original Edgebreaker decoding [21] and the need for the zipping procedure used by the Wrap&zip decoding [22]. Furthermore, for meshes with handles, our scheme eliminates the need for multiple traversals of the CLERS string and/or the mesh triangles during both encoding and decoding.

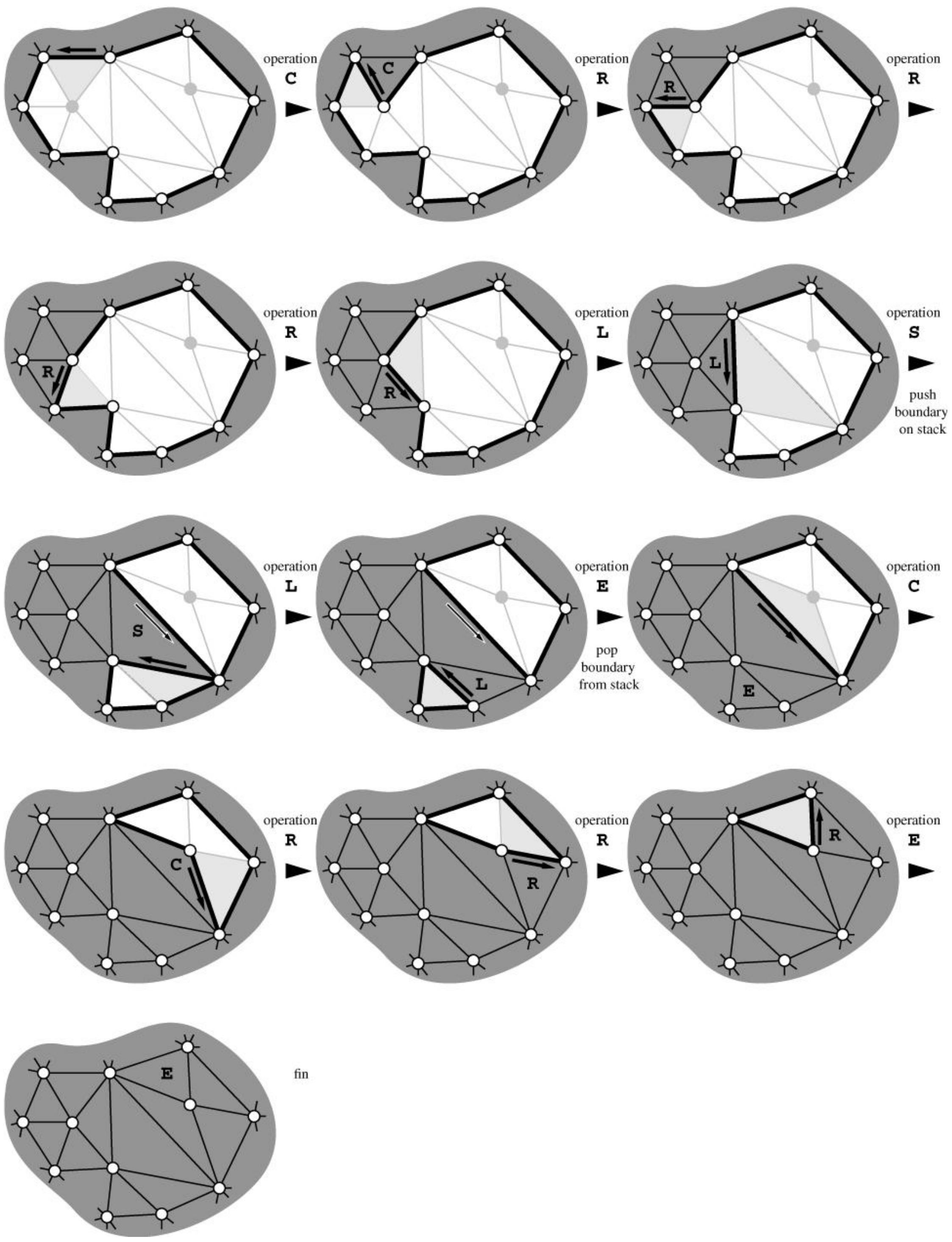


Figure 4.11: An example of the final twelve operations of *Edgebreaker encoding* a mesh.

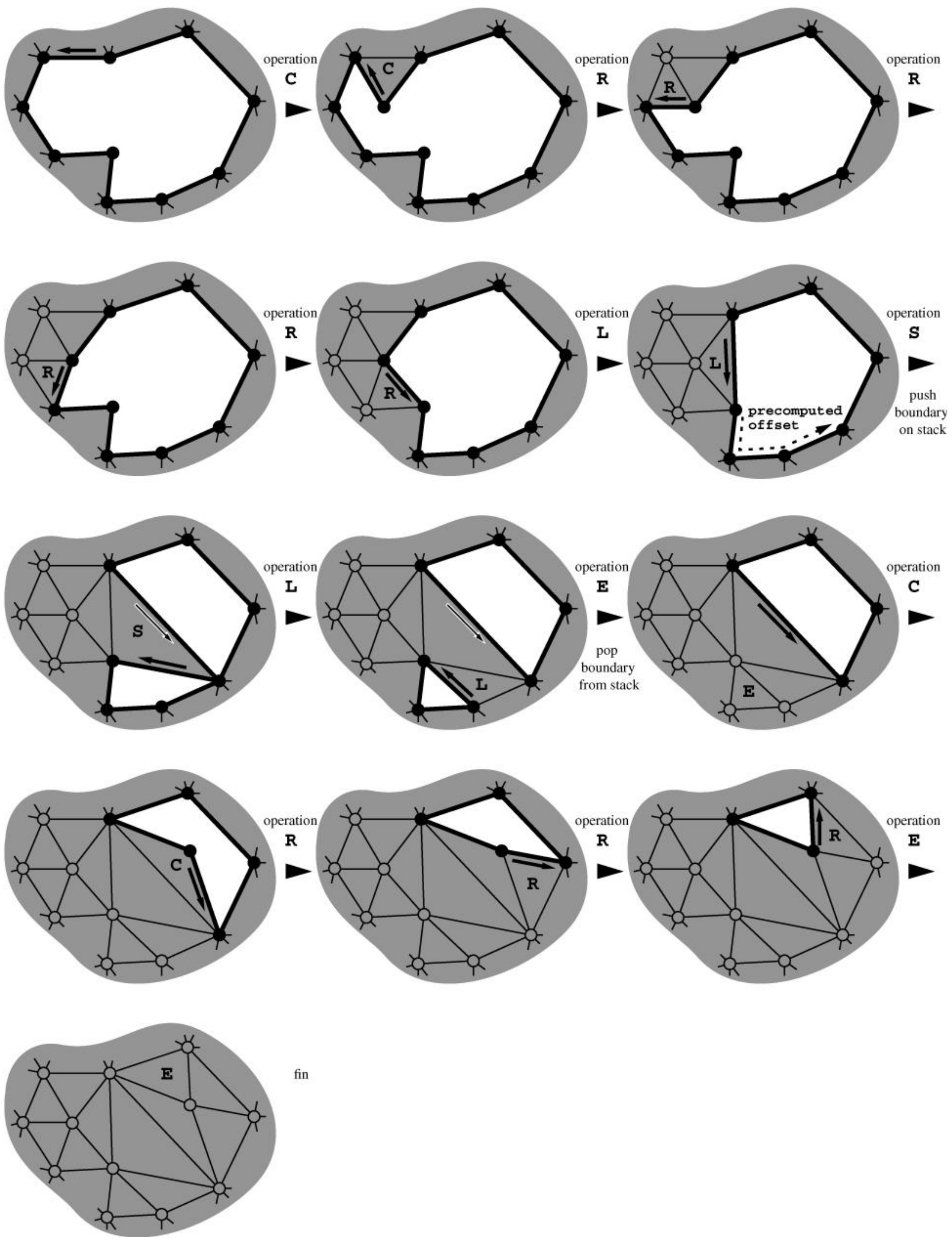


Figure 4.12: An example of the final twelve operations of *Edgebreaker decoding* a mesh.

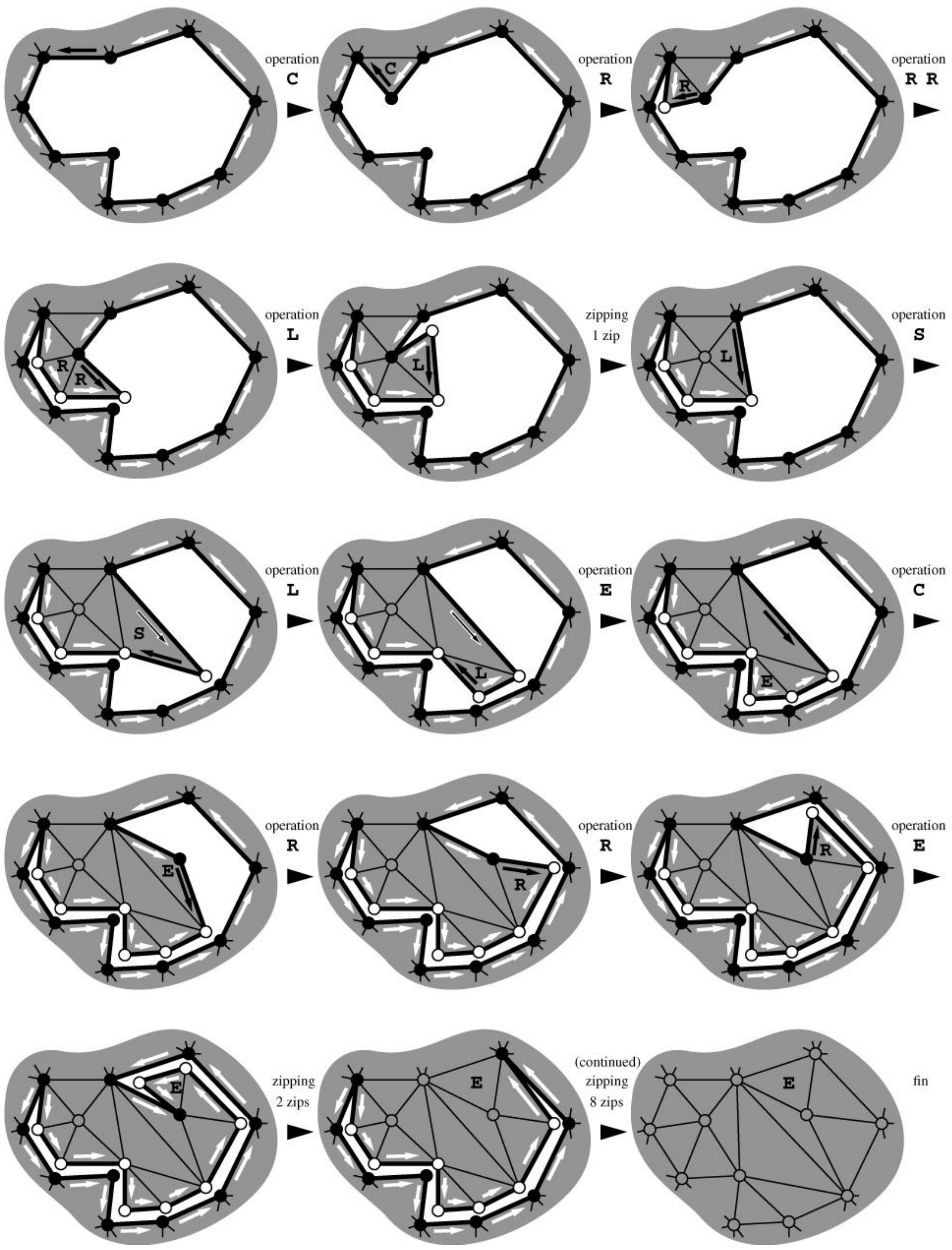


Figure 4.13: An example of the final twelve operations of *Wrap&zip decoding* a mesh.

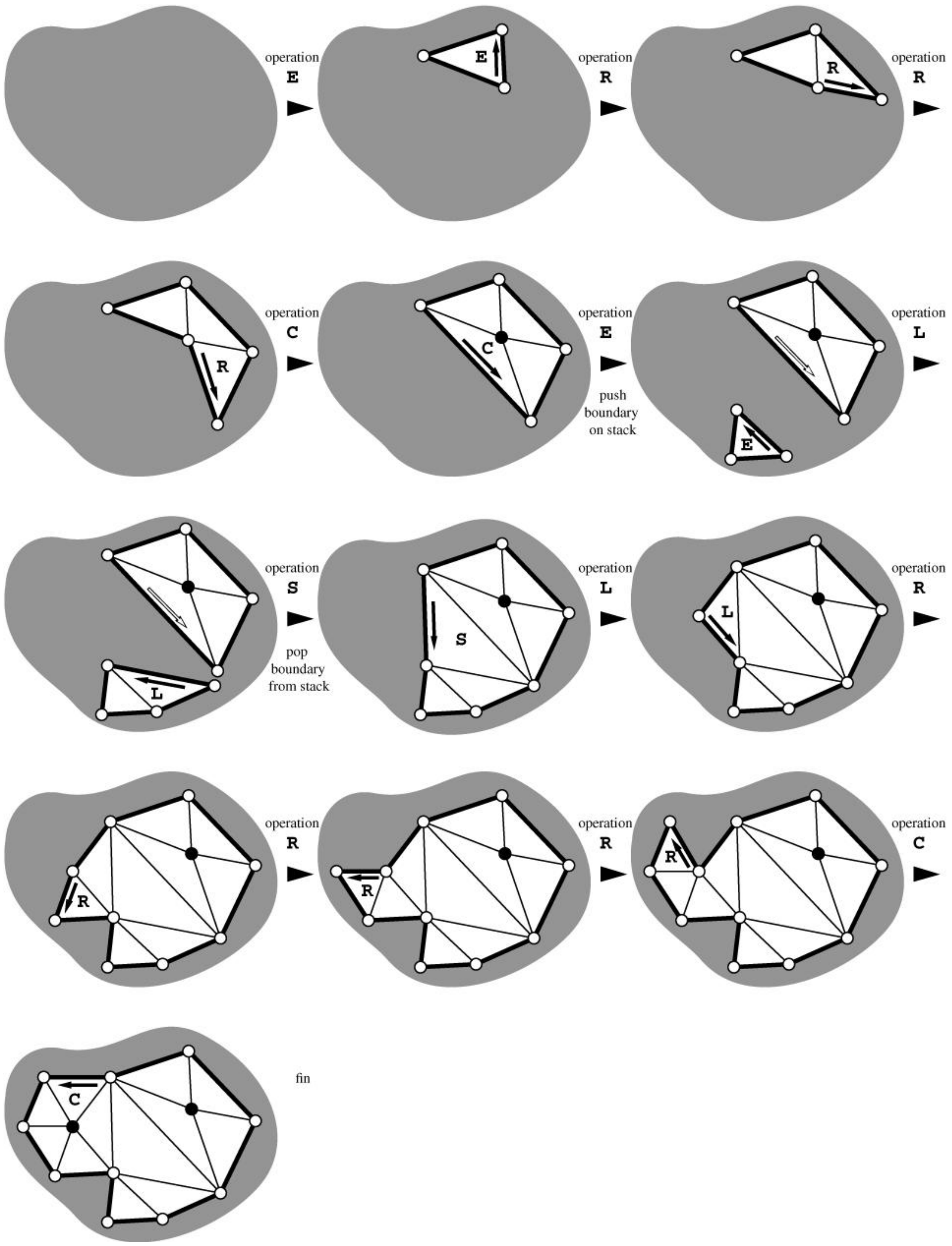


Figure 4.14: An example of the first twelve operations of *Spirale Reversi decoding* a mesh.

Bibliography

[1] H. Bruggesser and P. Mani. Shellable decompositions of cells and spheres. *Math. Scand.*, 29:197–205, 1971.

[2] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks, and W. V. Wright. Simplification envelopes. In *SIGGRAPH'96 Conference Proceedings*, pages 119–128, 1996.

[3] L. de Floriani, P. Magillo, and E. Puppo. A simple and efficient sequential encoding for triangle meshes. In *Proceedings of 15th European Workshop on Computational Geometry*, pages 129–133, 1999.

[4] M. Denny and C. Sohler. Encoding a triangulation as a permutation of its point set. In *Proceedings of 9th Canadian Conference on Computational Geometry*, pages 39–43, 1997.

[5] H. Edelsbrunner. private communication.

[6] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH'97 Conference Proceedings*, pages 209–216, 1997.

[7] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In *SIGGRAPH'98 Conference Proceedings*, pages 133–140, 1998.

[8] H. Hoppe. Progressive meshes. In *SIGGRAPH'96 Conference Proceedings*, pages 99–108, 1996.

[9] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *SIGGRAPH'93 Conference Proceedings*, pages 19–26, 1993.

[10] M. Isenburg. Triangle fixer: Compressing triangle meshes. In *preparation*, 2000.

[11] M. Isenburg. Triangle strip compression. In *preparation*, 2000.

[12] M. Isenburg and J. Snoeyink. Mesh collapse compression. In *Proceedings of SIBGRAPI'99 - 12th Brazilian Symposium on Computer Graphics and Image Processing*, pages 27–28, 1999.

[13] M. Isenburg and J. Snoeyink. Mesh collapse compression video. In *Proceedings of SCG'99 - 15th ACM Symposium on Computational Geometry*, pages 419–420, 1999.

[14] M. Isenburg and J. Snoeyink. Spirale reversi: Reverse decoding of the edgebreaker encoding. In *UBC Technical Report TR-99-08*, 1999.

[15] M. Isenburg and J. Snoeyink. Face fixer: Compressing polygon meshes with properties. In *preparation*, 2000.

[16] K. Keeler and J. Westbrook. Short encodings of planar graphs and maps. In *Discrete Applied Mathematics*, pages 239–252, 1995.

[17] D. King and J. Rossignac. Guaranteed 3.67v bit encoding of planar triangle graphs. In *Proceedings of 11th Canadian Conference on Computational Geometry*, pages 146–149, 1999.

[18] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal of Computing*, 12(1):28–35, 1983.

[19] D. G. Kirkpatrick. Establishing order in planar subdivisions. *Discrete Computational Geometry*, 3:267–280, 1988.

[20] L. Kobbelt, S. Campagne, and H. P. Seidel. A general framework for mesh decimation. In *GI'98 Conference Proceedings*, pages 43–50, 1998.

[21] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1), 1999.

[22] J. Rossignac and A. Szymczak. Wrap&zip: Linear decoding of planar triangle graphs. *The Journal of Computational Geometry, Theory and Applications*, 1999.

[23] J. Snoeyink and M. van Kreveld. Linear-time reconstruction of Delaunay triangulations with applications. In *Proceedings of 5th European Symposium on Algorithms*, pages 459–471, 1997.

[24] G. Taubin, A. Guéziec, W. Horn, and F. Lazarus. Progressive forest split compression. In *SIGGRAPH'98 Conference Proceedings*, pages 123–132, 1998.

[25] G. Taubin and J. Rossignac. Geometric compression through topological surgery. In *ACM Transactions on Graphics*, pages 17(2):84–115, 1998.

[26] C. Touma and C. Gotsman. Triangle mesh compression. In *GI'98 Conference Proceedings*, pages 26–34, 1998.

[27] G. Turan. Succinct representations of graphs. *Discrete Applied Mathematics*, 8:289–294, 1984.

[28] W.T. Tutte. A cencus of planar triangulations. *Canadian Journal of Mathematics*, 14:21–38, 1962.