Mesh Collapse Compression

Martin Isenburg Jack Snoeyink

Department of Computer Science
University of British Columbia
{isenburg | snoeyink}@cs.ubc.ca

November 15, 1999

Technical Report: TR-99-10

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 report we present a novel algorithm for encoding the topology of triangular meshes. Our encoding algorithm is based on the edge contract operation, which has been used extensively in the area of mesh simplification, but not for efficient mesh topology compression. We perform 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. Surfaces of higher genus are encoded at the expense of a few extra bits per handle.

Key words: Triangle mesh compression, topology encoding, edge contraction.

1 Introduction

Efficiently encoding the topology of triangular meshes has recently been the subject of intense study [15, 24, 23, 25, 20, 3, 9] 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 report introduces *Mesh Collapse Compression* (mc-compression), a new algorithm for encoding the topology of triangular meshes. An extended abstract describing this algorithm has appeared in [11] and a video illustrating this work in its early stages can be found in [12].

Our topology encoding algorithm is based on the edge contract operation, which has received attention in the computer graphics community. Hoppe [8, 7] made extensive use of the edge contract operation (calling it “edge collapse”) for topology preserving mesh simplification and others followed his approach [2, 5, 19]. But we are not aware of any encoding technique that uses the edge contract operation for efficient mesh topology compression.

In the next section we will define what triangle meshes are, as their efficient encoding is the meat of this report. In Section 3 we will give a comparative survey of previous work in this area. In Section 4 we will introduce our new encoding algorithm for the case of simple triangle meshes. In Section 5 we will show how our scheme extends to general triangle meshes with boundary, holes, and handles. Finally we will summarize our contributions.

2 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 boundary.

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 tri-

angle 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 [18]. 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 [27] 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.

3 Previous work

All efficient compression schemes that have been recently proposed for encoding triangle mesh connectivity [26, 15, 24, 25, 6, 20, 3, 9] 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 [20]). 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 [24, 25, 6, 20, 9]. Mesh Collapse Compression follows this pattern.

A different approach to topology encoding was presented by Snoeyink and van Kreveld for Delaunay triangulations [22]. Their scheme uses results by Kirkpatrick [17] 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 [26] 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 [15] 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 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 [24] 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 [25] 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 decoding time and bit-rate.

Gumhold and Strasser [6] introduce a compressed representation for triangle meshes that is closely related to the Edgebreaker method [20]. 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 [25]. The required offset value leads to the same non-linear behaviour, since $\log_2 v$ bits are required to encode it. However, it allows encoding and decoding to run practically in parallel, making it possible to *stream* a mesh across a network. 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 [20] was independently developed and gives the best guaranteed bit-rates for triangle mesh connectivity. Since the original paper many improvements have been reported [16, 21, 13]. 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 [6].

Improving on the original Edgebreaker decoding scheme [20], which has non-linear time complexity, Rossignac and Szymczak introduced the Wrap&zip decoding [21] that decodes simple meshes in provably linear time. However, this technique requires an overhead in data structure as each mesh edge of the vertex spanning tree initially appears twice. Also for meshes with handles the Wrap&zip scheme needs to perform multiple traversals of all mesh triangles. The Spirale Reversi decoding scheme [13] also achieves strict linear decoding time, but does not require any additional data structure. It decodes Edgebreaker encoded meshes with a single inverse traversal of the CLERS operation sequence.

The work by King and Rossignac [16] 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 [27].

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 [20] or the "new vertex", "connect forward", and "connect backward" operations of Gumhold and Strasser's scheme [6].

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 [25] as vertex based, Gumhold and Strasser [6], Edgebreaker [20], De Floriani et al. [3] as triangle based and Topologycal Surgery [24] as vertex *and* triangle based. Isenburg will soon present Triangle Fixer [9], a truly edge-based algorithm.

Using a simple fixed-bit encoding scheme, Triangle Fixer gives a 6 bpv guaranteed and a 3.9 to 4.2 bpv expected bit-rate that go down to 0.9 to 2.7 bpv with arithmetic coding. It has relatively simple extensions towards triangle strip compression [10] and polygon mesh compression [14], which make it especially interesting.

The next section introduces Mesh Collapse Compression, a compression scheme that falls into the vertex-based category. This method is most closely related to that of Touma and Gotsman [25]. 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.

4 Mesh Collapse Compression

In this section we introduce the Mesh Collapse Compression algorithm, prove its correctness, and present results on various example meshes. Restrictions on the mesh topology that are imposed here for the sake of simplicity are lifted in the next section.

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.

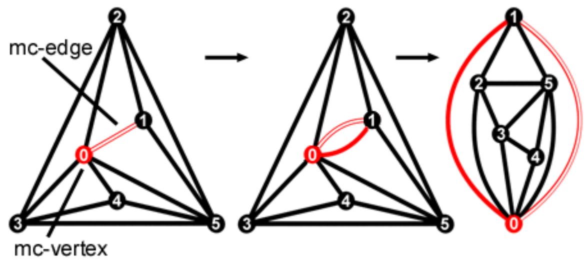

Figure 1: Cutting and opening the mc-edge turns the mesh into a digon.

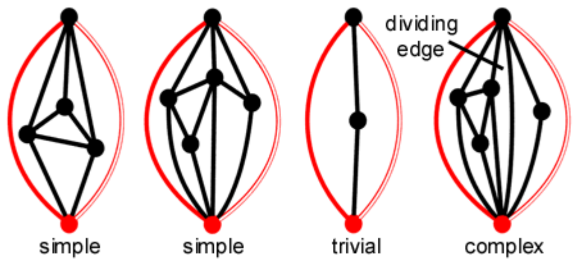

Figure 2: Two simple, one trivial, and one complex digon.

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 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 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.

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*.

4.1 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 3.

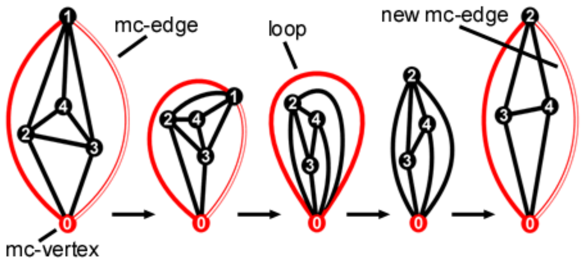

Figure 3: An illustration of the mc-contract operation.

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.

4.2 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 4 is an illustration of the mc-divide operation.

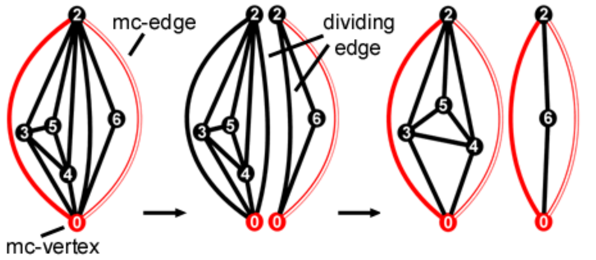

Figure 4: An illustration of the mc-divide operation.

4.3 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.

4.4 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 5. Finally the operation updates the mc-edge.

4.5 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

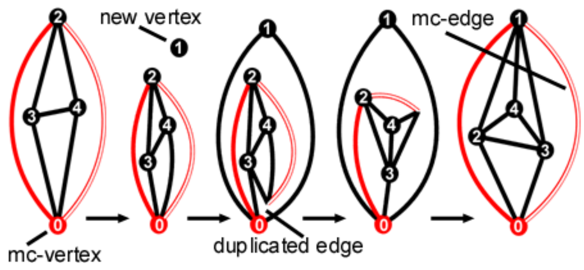

Figure 5: An illustration of the mc-expand operation.

will have $d + 1$ dividing edges. For an illustration of the mc-join operation read Figure 4 from right to left.

4.6 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 6 using digons (left) and using triangulations (right).

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 7.

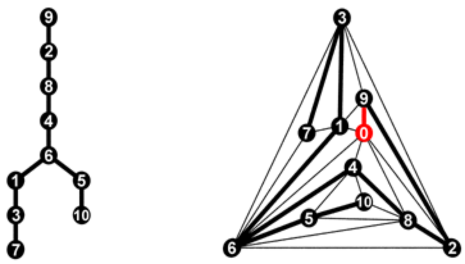

Figure 7: The mc-tree and its embedding in the mesh.

4.7 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);
}
```

4.8 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.

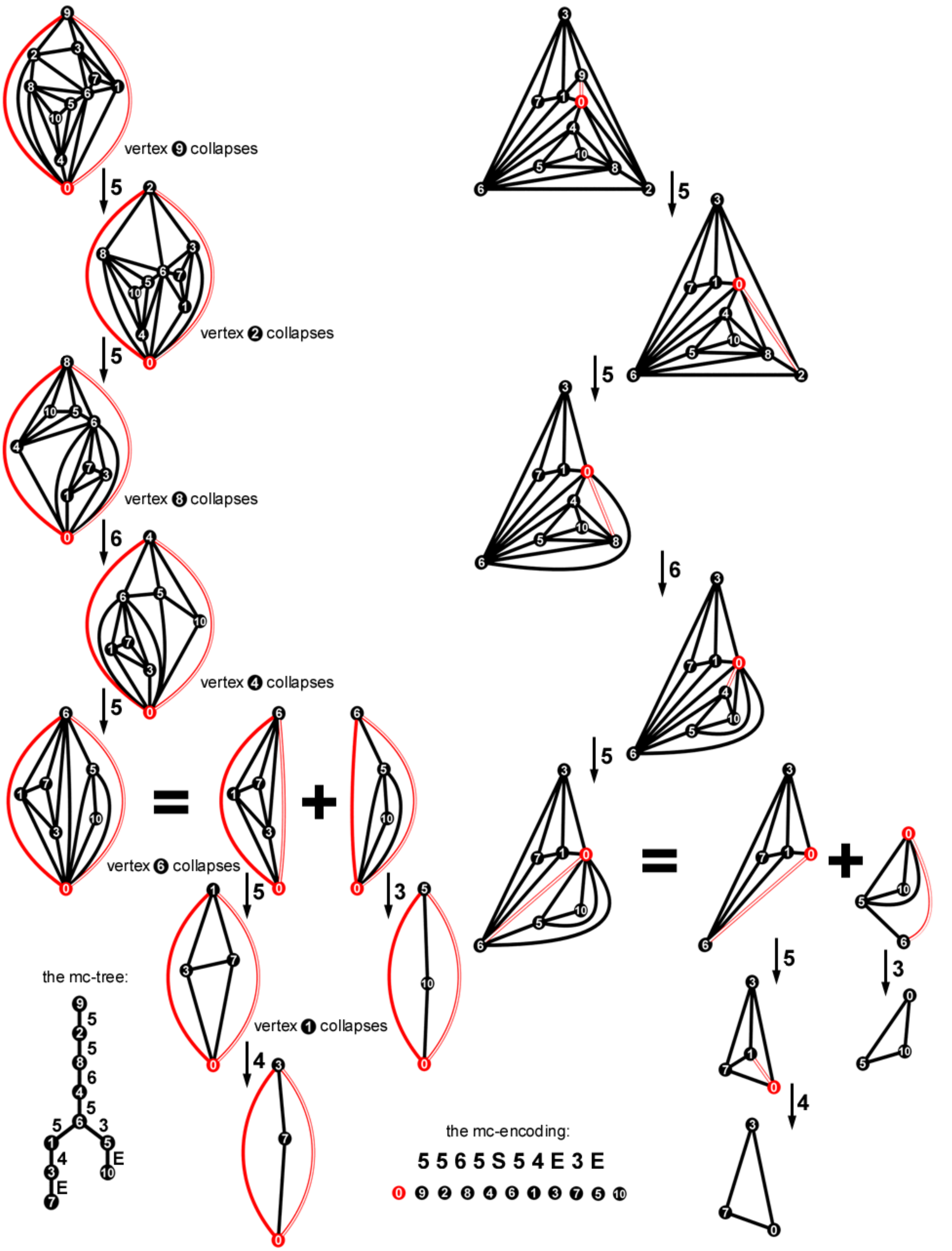

Figure 6: A small mesh is mc-compressed with seven mc-contract and one mc-divide operations.

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

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:

$$\begin{array}{lll}
\text{Indct. Base:} & c(3) = 0 & \\
\text{Indct. Assumption:} & c(k) = k-3 & \text{for } 3 \leq k < v \\
\text{Proof with axiom 2:} & c(v) = 1 + c(v-1) & \text{for } v > 3 \\
& \quad = 1 + (v-1) - 3 & \\
& \quad = v - 3 & \text{q.e.d.} \\
\text{Proof with axiom 3:} & c(v) = 1 + c(w) + c(v-w+2) & \\
& & \text{for } v > 3 \\
& \quad = 1 + w - 3 + v - w + 2 - 3 & \\
& \quad = v - 3 & \text{q.e.d.}
\end{array}$$

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.

The results of mc-compressing various example meshes are summarized in Table 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 [25] for similar code sequences promises even more compact encodings.

5 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.

5.1 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.

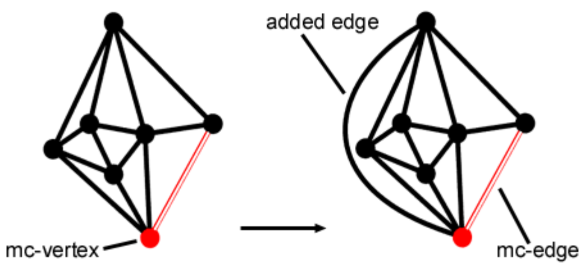

Figure 8: Patching a boundary with an additional edge.

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 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 9. These dummy vertices are marked and can be removed in a corresponding postprocessing step.

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

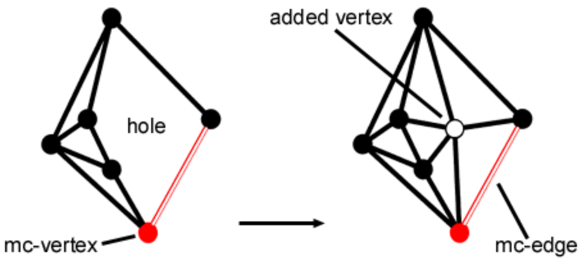

Figure 9: Patching a hole with a dummy vertex.

5.2 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 10.

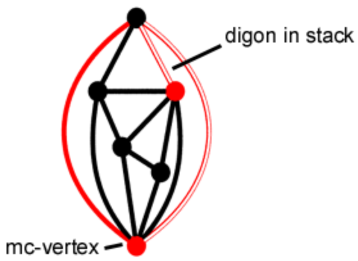

Figure 10: Encoding handles in the mesh.

6 Summary and Acknowledgements

We presented a novel encoding scheme for mesh topology. Our algorithm is simpler than approaches by [20, 25, 3] and produces a code sequence similar to [25]. 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.

This work has been supported by NSERC, IRIS, and a UBC Graduate Fellowship. Special thanks to Gene Lee for his RASP tools and his technical support during the making of the mc-video.

7 References

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

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

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

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

[9] M. Isenburg. Triangle fixer: Edge-based connectivity encoding. In *Proceedings of 16th European Workshop on Computational Geometry*, pages 24–29, 2000.

[10] M. Isenburg. Triangle strip compression. In *submitted*, 2000.

[11] 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.

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

[13] M. Isenburg and J. Snoeyink. Spirale reversi: Reverse decoding of the Edgebreaker encoding. Technical Report TR–99–08, Department of Computer Science, University of British Columbia, sep 1999.

[14] M. Isenburg and J. Snoeyink. Face Fixer: Compressing polygon meshes with properties. In *submitted*, 2000.

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

[16] 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.

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

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

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

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

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

[22] 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.

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

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

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

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

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