ELSEVIER

Computational Geometry 14 (1999) 137–166

Computational Geometry
Theory and Applications

www. elsevier.nl/locate/comgeo

Efficient compression of non-manifold polygonal meshes ☆

André Guéziec [a,*], Frank Bossen [b,1], Gabriel Taubin [c,2], Claudio Silva [d,3]

[a] *Multigen Paradigm, 550 S. Winchester Bvd., Suite 500, San Jose, CA 95128, USA*
[b] *Signal Processing Lab, EPFL, 1015 Lausanne, Switzerland*
[c] *IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA*
[d] *AT&T Labs-Research, 180 Park Ave, P.O. Box 971, Florham Park, NJ 07932, USA*

Communicated by J.-R. Sack

Abstract

We present a method for compressing non-manifold polygonal meshes, i.e., polygonal meshes with singularities, which occur very frequently in the real-world. Most efficient polygonal compression methods currently available are restricted to a manifold mesh: they require converting a non-manifold mesh to a manifold mesh, and fail to retrieve the original model connectivity after decompression.

The present method works by converting the original model to a manifold model, encoding the manifold model using an existing mesh compression technique, and clustering, or *stitching* together during the decompression process vertices that were duplicated earlier to faithfully recover the original connectivity. This paper focuses on efficiently encoding and decoding the stitching information. Using a naive method, the stitching information would incur a prohibitive cost, while our methods guarantee a worst case cost of $O(\log m)$ bits per vertex replication, where m is the number of non-manifold vertices. Furthermore, when exploiting the adjacency between vertex replications, many replications can be encoded with an insignificant cost.

By interleaving the connectivity, stitching information, geometry and properties, we can avoid encoding repeated vertices (and properties bound to vertices) multiple times; thus a reduction of the size of the bit-stream of about 10% is obtained compared with encoding the model as a manifold.

Keywords: Polygonal mesh; Geometry compression; Non-manifold; Stitching

1. Introduction

Three-dimensional polygonal meshes are used pervasively in manufacturing, architectural, Geographic Information Systems, warfare simulation, medical imaging, robotics, and entertainment industries. In

☆ This research was conducted while all authors were with IBM.

* Corresponding author. E-mail: gueziec@computer.org
[1] E-mail: frank.bossen@epfl.ch
[2] E-mail: taubin@us.ibm.com
[3] E-mail: csilva@research.att.com

0925-7721/99/$ – see front matter
PII: S0925-7721(99)00027-9

particular, polygons (especially, triangles) are required for generating three-dimensional renderings using available three-dimensional computer graphics architectures.

The sizes of such meshes have been steadily increasing, and there is no indication that this trend will change. For instance, a polygonal model representing a Boeing 777 airplane contains on the order of 1 Billion polygons, excluding polygons associated with the rivet models [2]. Other particularly large meshes include architectural models. Geometry compression deals with the compression of polygonal meshes for transmission and storage.

Many real-world polygonal meshes are *non-manifold*, that is, contain topological singularities, (e.g., edges shared by more than two triangles) [4]. In fact, on a database of 303 meshes used for MPEG-4 core experiments and obtained on the Web (notably at the www.ocnus.com site), we discovered that more than half of the meshes (162 precisely) were non-manifolds.

As discussed in Section 2, most of the methods currently available for geometry compression require a manifold connectivity [5]. Meshes can be converted from a non-manifold connectivity to a manifold connectivity, but then the original connectivity is lost. At the time this paper is written, we are aware of three publications addressing connectivity-preserving non-manifold mesh compression [1,13,23].

We describe a method for compressing non-manifold polygonal meshes and recovering their exact connectivity after decompression. Excerpts of this paper were published in [7]. Our method compares in compression efficiency and speed with the most efficient manifold-mesh compression methods, thus extending [9,12–14,22], and even allows some savings by avoiding duplicate encodings of vertex coordinates and properties, as reported in Section 12.

Although it may be questionable to assign unique texture coordinates or normals to a non-manifold vertex, we do not make any assumption of integrity or consistency in the original data, but only strive to encode and transmit it integrally. Specifically, our method will process any polygonal mesh represented as a VRML IndexedFaceSet [21], or as indicated in the MPEG-4 specifications [11].

Throughout this paper, we will demonstrate the algorithms using a simple mesh example composed of two tetrahedra sharing a common face. This mesh has seven faces and five vertices as illustrated in Fig. 1. Three of these vertices are non-manifold vertices (vertices 0, 1 and 2 in Fig. 2). See Appendix A for a precise definition of a non-manifold vertex.

The present method works by converting the original mesh to a set of manifold meshes as exemplified by Fig. 2, encoding the manifold meshes using an existing mesh compression technique, and clustering together during the decompression process the vertices that were duplicated earlier to faithfully recover the original connectivity (see Fig. 3). To convert a non-manifold mesh to a manifold mesh, we duplicate non-manifold vertices as described in detail in [8] and briefly recalled in Section 3. Other methods could be used as well. During the conversion process, we record the information necessary to cluster the vertices that were duplicated. We call this information a *vertex clustering*.

1.1. Motivation

The basic idea in this paper is to encode both the manifold meshes and vertex clustering as a substitute for the non-manifold mesh.

[4] Specifically, a manifold polygonal mesh is such that each vertex of the mesh is a manifold vertex, i.e., the neighborhood of each vertex can be continuously deformed to a disk (to a half disk at the boundary). See Appendix A for details.

[5] We call *connectivity* the graph of polygon-vertex incidences.

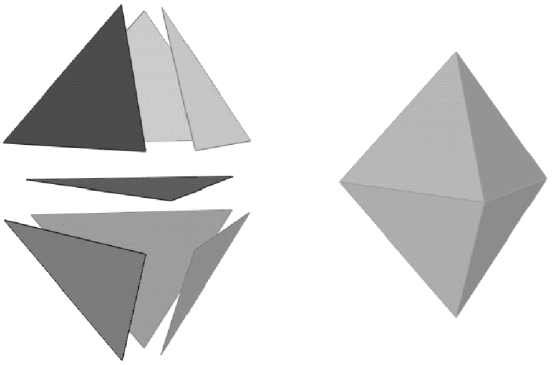


Fig. 1. The seven faces (a) composing a non-manifold mesh (b) that we will use as an example throughout this paper.

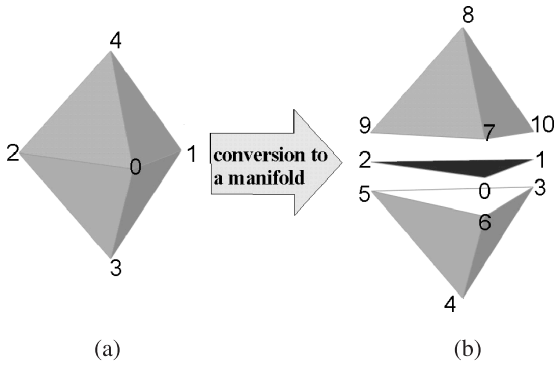


Fig. 2. Converting a non-manifold mesh (a) to a manifold mesh (b) by cutting through non-manifold vertices, i.e., replicating non-manifold vertices (vertices 0, 1 and 2) [8].

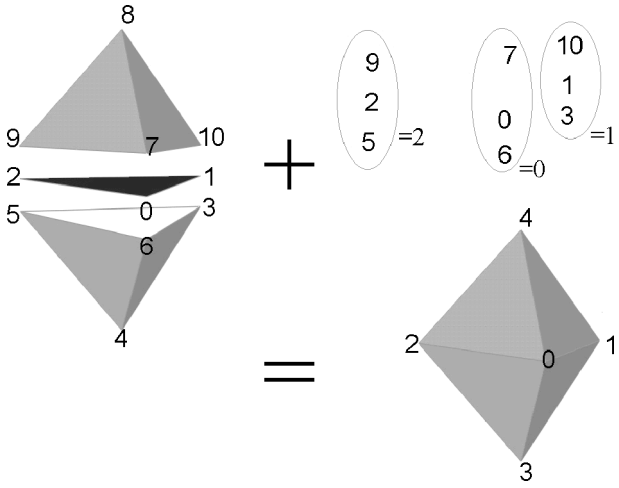


Fig. 3. To encode a non-manifold mesh, we encode a corresponding manifold mesh, and which vertices must be clustered to recover the original connectivity.

A naive approach for doing this would operate by transmitting the manifold meshes using existing mesh compression methods, and then transmitting the specification of pairs of indices to vertices that must be clustered. This approach has two important drawbacks: firstly, the (x, y, z) coordinates and properties of vertices that were duplicated during the non-manifold to manifold conversion will be encoded and transmitted multiple times. The cost associated with this duplication depends on the input mesh: on a sample of 14 meshes that we used for experiments, we estimated this cost to be in average on the order of 10% of the total bit-stream size (see Section 12). Secondly, the specification of pairs of vertex indices is costly: $2 \log n$ bits for each pair, wherein n is the number of vertices in the mesh after non-manifold to manifold conversion.

Table 1
To encode the information necessary to cluster some vertices and recover the polygon–vertices incidence relationship of the original model, one simple approach (which we are not using, but explain for comparison) would be to transmit a table such as this, with one row per cluster. This table relates to the example of Figs. 1–3

Number of vertices in cluster	Vertex references		
3	0	6	7
3	2	5	9
3	1	3	10

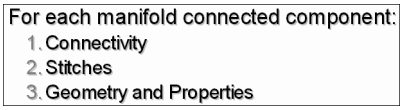


Fig. 4. Overall compressed syntax for a non-manifold mesh.

Another approach consists of devising special codes to indicate which vertices are copies of previously transmitted vertices, specifying which vertex they should be clustered to. This method avoids duplicate transmission of vertex coordinates and properties, but incurs a cost of $\log n$ bits for each vertex that was previously transmitted.

A refined version of this approach would transmit a table with one row per cluster, as in Table 1. We denote by m the number of clusters. (If each non-manifold vertex must be replicated during the non-manifold-to-manifold mesh conversion process as discussed in Section 3, m is the same as the number of non-manifold vertices in the original mesh.) For instance, $m = 3$ in Fig. 2. Each entry of the table would indicate the number of vertices in the cluster and their references: this is shown in Table 1 for the example that we are using. If the total number of vertex replications is r ($r = 9$ in Fig. 2), and the maximum number of vertices in one cluster is c, the total cost of this approach is $m \log c + r \log n$.

In this paper, we improve considerably upon this second method, to incur a worst-case cost of at most $\log m$ bits for each vertex that was previously transmitted, where m is the number of clusters, which is typically much smaller than n. Moreover, this is a worst-case cost, as we will show in Section 5 that for many vertices, the clustering information is implicit, and requires zero bits to encode. This is achieved using the concept of *stitches*, which are defined and fully described in Section 5.

1.2. Representation for compression

The mesh is compressed as indicated in Fig. 4. For each manifold connected component, the connectivity is encoded, followed with optional *stitches*, and geometry [6] and properties. Stitches are used to recover the vertex clustering within the current component and between vertices of the current

[6] The vertex coordinates.

component and previous components. In this way, for each cluster, geometry and properties are only encoded and decoded for the vertex of the cluster that is encountered first (in decoder order of traversal).

1.3. Overview

In Section 2 we review related work. We review in Section 3 an algorithm for producing manifold meshes starting with a non-manifold mesh. Section 4 describes a method for compressing manifold meshes. Sections 5–8 develop methods for representing a vertex clustering using stitches. In Section 9 we give details on the encoding process and describe a proposed bit-stream syntax for stitches. In Section 10 we study the decoding process. Our algorithms are analyzed in Section 11. In Section 12 we provide compression results on a variety of non-manifold meshes.

2. Related work

2.1. Non-manifold mesh compression

Connectivity-preserving non-manifold mesh compression algorithms were proposed by Popovic and Hoppe [13] and Bajaj et al. [1]. Hoppe's Progressive Meshes [10] use a base mesh and a series of vertex insertions (specifically, inverted edge contractions) to represent a manifold mesh. While the main functionality is progressive transmission, the encoding is fairly compact, using 30–50 bits per vertex with arithmetic coding [10]. This method was extended in [13] to represent arbitrary simplicial complexes, manifold or not, using about 50 bits per vertex (asymptotically the cost of this method is $O(n \log n)$ bits, n being the number of vertices). This method of [13] works by devising special codes to encode all possible manifold or non-manifold attachments of a new vertex (and sustaining edges and triangles) to an existing mesh. A code must be supplied for each vertex that is encoded. Our present method improves upon [13] by achieving significantly smaller bit-rates (about 10 bits per vertex or so) and reducing encoding time (admittedly, an off-line process) by more than four orders of magnitude (without levels-of-detail).

Bajaj et al.'s "DAG of rings" mesh compression approach [1] partitions meshes in vertex and triangle layers that can represent a non-manifold mesh. A vertex layer is a set of vertices with the same topological distance to an origin vertex. A vertex layer is a graph that may contain non-manifold vertices, which correspond to branching points. Encoding the various branches requires indices that are local to the vertex layer. In this paper, we encode the same information by indexing among a subset of the m non-manifold vertices (those present in a stack). With the variable-length method described in Section 7, we obtain additional savings by exploiting the adjacency between non-manifold vertices.

Another advantage of our approach is that we define a compressed syntax that can be used to cluster vertices. This syntax can be used for encoding some changes of topology (such as aggregating components) in addition to representing singularities.

Abadjev et al. [23] use a technique related to [10,13], and introduce a hierarchical block structure in the file format for parallel streaming of texture and geometry.

2.2. Manifold-mesh compression

For completeness, we now discuss previous work on compressing manifold meshes, which is related to our approach of Section 4.

Deering [5] introduced geometry compression methods, originally to alleviate 3D graphics rendering limitations due to a bottleneck in the transmission of information to the graphics hardware (in the bus). His method uses vertex and normal quantization, and exploits a mesh buffer to reuse a number of vertices recently visited and avoid re-sending them. Deering's work fostered research on 3D mesh compression for other applications. Chow [3] extended [5] with efficient generalized-triangle-strip building strategies.

The Topological Surgery single-resolution mesh compression method of Taubin, Rossignac et al. [19, 20] represents a connected component of a manifold mesh as a tree of polygons (which are each temporarily decomposed into triangles during encoding and recovered after decoding). The tree is decomposed into runs, whose connectivity can be encoded at a very low cost. To recover the connectivity and topology, this tree is completed with a vertex tree, providing information to merge triangle edges. The method of [19] also encodes the vertex coordinates (geometry) and all property bindings defined in VRML'97 [21].

Touma and Gotsman [22] traverse a triangular (or polygonal) mesh and remove one triangle at a time, recording vertex *valences* [7] as they go and recording triangles for which a boundary is split in two as a separate case.

Gumhold and Strasser [9] and Rossignac [14] concentrate on encoding the mesh connectivity. They use mesh traversal techniques similar to [22], but instead of recording vertex valences, consider more cases depending on whether triangles adjacent to the triangle that is being removed have already been visited. Another relevant work for connectivity compression is by Denny and Sohler [6].

Li and Kuo's [12] "dual graph" approach traverses polygons of a mesh in a breadth-first fashion, and uses special codes to merge nearby (topologically close) polygons (serving the same purpose as the vertex graph in the approach of [19]) and special commands to merge topologically distant polygons (to represent a general connectivity-not only a disk).

3. Cutting a non-manifold mesh to produce manifold meshes

We briefly recall here the method of Guéziec et al. [8] that we are using. For each edge of the polygonal mesh, we determine whether the edge is singular (has three or more incident faces) or regular (with two incident faces). Edges for which incident faces are inconsistently oriented are also considered to be singular for the purpose of this process of converting a non-manifold to a manifold. For each *singular* vertex of the polygonal mesh, the number of connected *fans* of polygons incident to it is determined. [8] For each connected fan of polygons, a copy of the singular vertex is created (thereby duplicating singular vertices). The resulting mesh is a manifold mesh. The correspondences between the new set of vertices comprising the new vertex copies and the old set of vertices comprising the singular vertices is recorded in a vertex clustering array. This process is illustrated in Fig. 2.

This method admits a number of variations that moderately alter the original mesh connectivity (without recovering it after decoding) in order to achieve a decreased size of the bit-stream: polygonal faces with repeated indices may be removed. Repeated faces (albeit with potentially different properties attached) may be removed. Finally, the number of singular edges may be reduced by first attempting to

[7] Number of incident polygons.

[8] A fan of polygons at a vertex is a set of polygons incident to a vertex and connected with regular edges. A singular vertex is simply a vertex with more than one incident fans.

invert the orientation of some faces in order to reduce the number of edges whose two incident faces are inconsistently oriented.

An interesting alternative for converting a non-manifold mesh to a manifold mesh by vertex replication was recently introduced by Rossignac and Cardoze [16]. Rossignac and Cardoze minimize the number of vertex replications when converting non-manifold solids to manifold solid representations. In the Rossignac and Cardoze method, an edge cannot be uniquely identified with a pair of vertices: for instance two edges (and four faces) can share the same two endpoints. In the method of Section 4 however, we have used the assumption that an edge could be uniquely identified using two vertices, which allows a simple representation and encoding for a vertex graph, and does not require to output a list of edges when decoding the compressed representation of the mesh: the edges are implicitly represented by the polygon-vertices incidence relation.

4. Compressing manifold meshes

The method described in this section extends the Topological Surgery method [19], and is explained in detail in [11]. In [19] the connectivity of the mesh is represented by a tree spanning the set of vertices, a simple polygon, and optionally a set of jump edges. To derive these data structures a vertex spanning tree is first constructed in the graph of the mesh and the mesh is cut through the edges of the tree. If the mesh has a simple topology, the result is a simple polygon. However, if the mesh has boundaries or a higher genus, additional cuts along jump edges are needed to obtain the simple polygon. This simple polygon is then represented by a triangle spanning tree and a marching pattern that indicates how neighboring triangles are connected to each other. The connectivity is then encoded as a vertex tree, a simple polygon and jump edges. In this paper the approach is slightly different. First, a triangle spanning tree is constructed. Then the set of all edges that are not cut by the triangle tree are gathered into a graph. This graph, called Vertex Graph, spans the set of vertices, and may have cycles. Cycles are caused by boundaries or handles (for higher genus models). The vertex graph, triangle tree, and marching pattern are sufficient to represent the connectivity of the mesh.

In [19], geometry and properties are coded differentially with respect to a prediction. This prediction is obtained by a linear combination of ancestors in the vertex tree. The weighting coefficients are chosen to globally minimize the residues, i.e., the difference between the prediction and the actual values. In this paper the principle of linear combination is preserved but the triangle tree is used instead of the vertex tree for determining the ancestors. Note that the "parallelogram prediction" [22] [9] is a special case of this scheme, and is achieved through the appropriate selection of the weighting coefficients.

Coding efficiency is further improved by the use of an efficient adaptive arithmetic coder [18]. Arithmetic coding is applied to all data, namely connectivity, geometry and properties.

Finally, the data is ordered so as to permit efficient decoding and on-the-fly rendering. The vertex graph and triangle tree are put first into the bit stream. The remaining data, i.e., marching pattern, geometry, and properties, is referred to as triangle data and is put next into the bit stream. It is organized on a per-triangle basis, following a depth-first traversal of the triangle tree. Therefore a new triangle

[9] Which extends a current triangle to form a parallelogram, with the new parallelogram vertex being used as a predictor.

may be rendered every time a few more bits, corresponding to the data attached to the triangle, are received.

5. Representing the vertex clustering using stitches

5.1. *Decoding order and father–child relationship:* v_father

The methods developed in the present paper rely on the availability of two main elements: (1) a decoding order for the mesh vertices, and (2) for the variable-length method described in Section 7, a father–child relationship between vertices allowing to define paths of vertices. We next suppose that this father–child relationship is recorded in an array called v_father, representing a function $\{1,\ldots,n\} \overset{\text{v\_father}}{\longrightarrow} \{1,\ldots,n\}$, where n is the number of vertices.

All of the manifold mesh compression methods reviewed in Section 2 can provide these two elements, which makes the methods of this paper widely applicable: an order in which vertices are decoded is always available, and a father–child relationship can by default be realized by each vertex pointing to the vertex decoded just before as its father (the first vertex being its own father).

Our assumption is that the information consigned in v_father is implicit (provided "for free" by the manifold mesh compression method), and requires no specific encoding. In the following we assume without loss of generality that vertices are enumerated in the decoder order of traversal. (If this is not the case, we can perform a permutation of the vertices.)

Preferably, the v_father array will contain additional information, that can be exploited by our algorithms. For instance, Fig. 5 shows v_father for the example of Fig. 2, obtained using the Topological Surgery method [11]. In the particular case of Topological Surgery, v_father represents a digraph whose nodes are mesh vertices, edges are mesh edges, and such that each node has out-degree one: v_father is a forest that also admits self-loops.

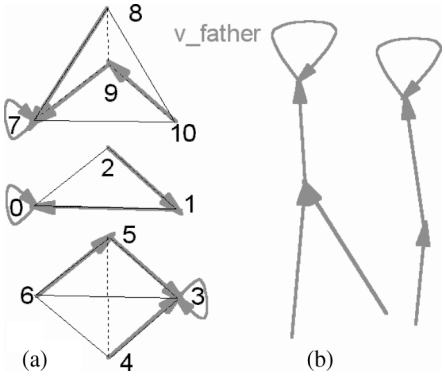


Fig. 5. (a) v_father (father–child) relationship for the example of Fig. 2, as generated by the Topological Surgery method. Since we are only concerned with topology, in this figure and the following figures, we are representing the mesh of Fig. 2 in wireframe mode, where dashed lines represent edges shared by two back-facing triangles. (b) In the particular case of Topological Surgery, v_father is a forest that also admits self-loops. In the following, we will omit to draw self-loops.

5.2. Stitches

A *stitch* is an operation that clusters a given number of vertices along two specified (directed) *paths* of vertices, wherein a path of vertices is defined by following a father–child relationship. There is no ambiguity in following this path, which corresponds to going towards the root of a tree. A stitch as defined above is also called a *forward* stitch. A stitch is specified by providing a pair of vertices and a length. The vertex clustering is accomplished by performing a series of stitches.

We may sometimes want to go down the father–child tree as opposed to up the tree (towards the root). This is a priori ambiguous, but the following definition removes the ambiguity: a *reverse* stitch works by starting with two vertices, following the path defined by the father–child relationship for the second vertex and storing all vertices along the path in a temporary structure, and clustering vertices along the path associated with the first vertex together with the stored vertices visited in reverse order.

We introduce two methods, called Stack-Based and Variable-Length, for representing (or decomposing) the vertex clustering in a series of stitches. These are two alternate methods that the encoder should choose from. The latter method presents more challenges but allows a much more compact encoding, as reported in Section 12. We define a bit-stream syntax that supports both possibilities, and it is not required that the encoder use the more advanced feature.

5.3. Vertex clustering array: v_cluster

Both the stack-based and variable-length methods take as input a vertex clustering array, which for convenience we denote by v_cluster ($\{1,\dots,n\} \overset{\text{v\_cluster}}{\longrightarrow} \{1,\dots,n\}$).

To access vertices through v_cluster, we propose the convention that v_cluster always indicate the vertex with the lowest decoder order: supposing that vertices 1 and 256 belong to different components but cluster to the same vertex, it is better to write $\text{v\_cluster}[1] = \text{v\_cluster}[256] = 1$ than $\text{v\_cluster}[1] = \text{v\_cluster}[256] = 256$. As the encoder and decoder build components gradually, at some point Vertex 1 will be a "physical" vertex of an existing component, while Vertex 256 will be in a yet-to-be-encoded component. Accessing Vertex 1 through Vertex 256 would increase code complexity.

The stack-based and variable-length methods are developed in the following sections. The variable-length method exploits the information in the v_father forest while the stack-based method does not. The stack-based method can be explained and implemented without requiring the notion of stitches: the stitches used for the stack-based method all have zero length, and associate a vertex of higher decoder order with a vertex of lower decoder order. However, stitches are at the core of the variable-length method, which is more efficient. It is useful to combine the two methods in a single framework and using a single compressed syntax.

6. Stack-based method

We wish to encode efficiently m clusters that affect r replicated vertices, while the rest of the $n - r$ vertices are not affected. (We can also say that their cluster size is 1.) The main idea behind this method is that provided that we can keep a stack of cluster representatives, only $\log m$ bits will be necessary for each of the r vertices to indicate which cluster they belong to. We can compress this information even

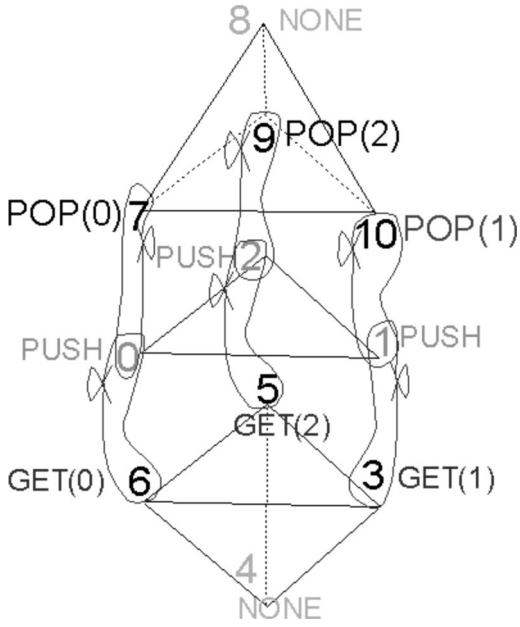


Fig. 6. Stack-based method applied to the example of Fig. 1. To represent stitches pictorially, we are using "lassos" and "wiggles".

further if only a portion of the m cluster representatives are present in a stack at a given time, by using as many indices as elements present in the stack.

For this purpose, we use a *stack-buffer*, similarly to Deering [5] and other manifold mesh compression modules (see [11]). A "stack" would only support "push" and "pop" operations. We denote by "stack-buffer" a data structure that supports the "get" operation as well as direct indexing into the stack for "get" and "pop" operations.

We push, get and pop in the stack-buffer the vertices that cluster together. Connected components can be computed for the vertex clustering, such that two vertices belong to the same component if they cluster to the same vertex. In the decoding order, we associate a *stitching command* to each vertex. If the size of the vertex's component is one, the command is NONE. (This is one of the $n - r$ unaffected vertices.) Otherwise, the command is either PUSH, or GET(v), or POP(v) depending on the decoding order of the vertices in a given component, where v is an index local to the stack-buffer. The vertex that is decoded first is associated with a PUSH; all subsequently decoded vertices are associated with a GET(v) except for the vertex decoded last, that is associated with a POP(v), whereby v is removed from the stack-buffer.

For the example of Fig. 1 we illustrate the association of commands to vertices in Fig. 6. Each vertex is labeled using its decoder order, and the corresponding command is displayed in the vicinity of the vertex.

In order to achieve this local indexing to the set of clusters and thus avoid incurring a $\log n$ cost when encoding vertex repetitions, we need to provide a code (stitching command) for each vertex: NONE, PUSH, GET or POP. As discussed in Section 9, the NONE command uses one bit, which may be further compressed using arithmetic coding [18] (when considering the sequence of commands). An alternative to this one-bit-per-vertex cost would be to provide a list of cluster representatives, each requiring a $\log n$ index to encode. We have not selected this approach.

In addition to providing lower bit rates, another advantage of our approach is that the command that is associated with each vertex may be used to decide whether or not to encode its coordinates and properties: geometry and properties may only be encoded for NONE and PUSH vertices. We can thus easily interleave connectivity and geometry in the bitstream, allowing incremental decoding and rendering (see Section 4). This can be done without the overhead of encoding a table of clusters.

One drawback of the stack-based method is that it requires to send one stitching command different from NONE (either PUSH, GET or POP) for each of the r vertices that are repeated (that cluster to a singular vertex). In the next section, we explain how the variable-length method exploits the situation when cluster members are adjacent in the v_father forest in order to replace as many PUSH, GET and POP commands as possible with a NONE command (that requires only one bit).

7. Variable-length method

7.1. Principle of the method

If we take a closer look at Fig. 5 we realize that the clusters (2,5) and (1,3) are such that 1 is the father of 2 and 3 is the father of 5. This is an example of a situation where we can add length to a stitch as defined in Section 5: stitch vertex 5 to vertex 2, with a stitch length of 1. A pictorial representation of this stitch is provided in Fig. 7(a). We will thus push 2 in the stack; Vertex 5 will be associated with a GET(2) command and a stitch length of 1. This will have the effect of fetching the fathers of 2 and 5, 1 and 3, and clustering them. (If the length was 2, we would fetch the fathers of 1 and 3 and cluster them as well, and so on.) The advantage of this is that vertex 1 and vertex 3 do not require a specific PUSH, GET, or POP command: we can associate them with a NONE command, which requires significantly fewer bits.

With a suitable v_father father–child relationship, we expect the above situation to occur frequently. In particular, in Topological Surgery, all edges of a mesh boundary except one belong to v_father. Since we expect a lot of stitching to occur along boundaries, the paths defined by v_father will be very valuable.

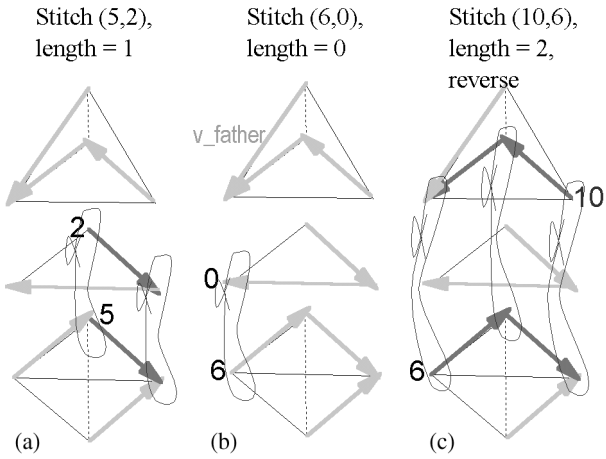


Fig. 7. Three stitches of variable length and direction encode the vertex clustering of Fig. 3: (a) length 1, (b) length 0, (c) length 2 and reverse direction.

Using the example of Fig. 3, we illustrate in Fig. 7 how variable length stitches can be used to represent the complete vertex clustering: three stitches are applied to represent v_cluster: one (forward) stitch of length 1 that was discussed above, one stitch of length zero (6, 0), and one reverse stitch of length 2 (10, 6). According to the definition provided in Section 5, the reverse stitch is performed as follows: we retrieve the father of 6, 5, and the father of 5, 3. We cluster 10 with 3, then the father of 10, 9, with 5 and, finally, the father of 9, 7, with 6.

In the remainder of this section, we explain how to discover such stitches from the knowledge of the v_cluster and v_father arrays.

7.2. Discovering the variable-length stitches

A good working hypothesis states that: the longer the stitches, the fewer the commands, and the smaller the bit-stream size. We propose a greedy method that operates as follows. We generate the longest stitch starting at each vertex, and we perform the stitches in order of decreasing length. The justification of this is that, assuming all stitches have the same direction, redundant encodings can be avoided. This is illustrated in Fig. 8: when a stitch is not extended to its full possible length (stitch 1), another stitch (stitch 2) could encode redundant information, unless it is broken up in smaller stitches. However, if all stitches are always extended to their full possible length, subsequent stitches may simply be shortened if necessary to avoid redundancy (instead of broken up).

We can thus safely apply all the (forward) stitches one after the other in order of decreasing length: for each stitch, we simply recompute its length appropriately (for instance, stitch 2 in Fig. 8 should be of length 1, and not 3). (When reverse stitches are introduced, however, the situation is more complex, as illustrated in Fig. 12 and discussed in Section 8.)

The method first computes for each vertex that clusters to a singular vertex the longest possible forward stitch starting at that vertex: a length and one or several candidate vertices to be stitched with are determined. As illustrated in Fig. 9(a), starting with a vertex v_0, $v_0 \in \{1, \ldots, n\}$, all other vertices in the same cluster are identified, and v_father is followed for all these vertices. From the vertices thus obtained, the method retains only those belonging to the same cluster as v_father$[v_0]$. This process is iterated until

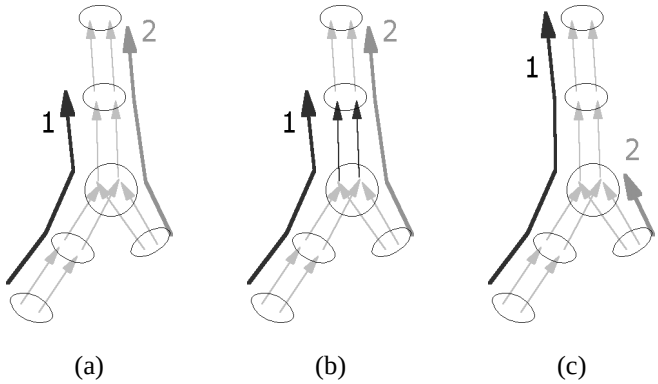


Fig. 8. Justification of the longest-stitch-first strategy: Supposing stitch 2 is performed after stitch 1 (a), the stitching information in light gray (b) will be encoded twice. (c) With our strategy this cannot happen, since stitch 1 can, and will, be prolonged to a stitch of length 4, and stitch 2 will be shortened to a length of 1.

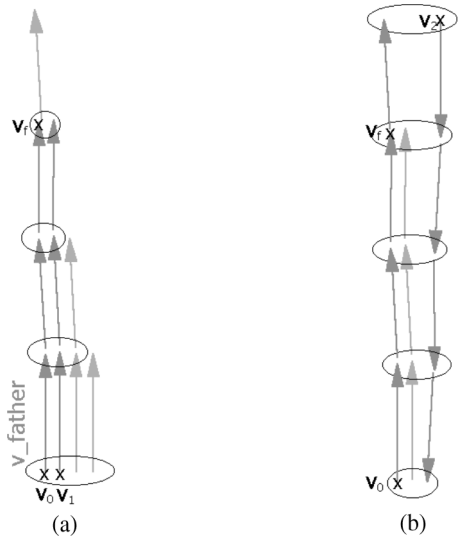


Fig. 9. Computing the longest possible stitch starting at a vertex v_0. Ovals indicate clusters. (a) Forward stitch of length 3 with v_1. (b) Backward stitch of length 4 with v_2.

the cluster contains a single vertex. The ancestors of vertices remaining in the previous iteration (v_f is the successor of v_0 ending the stitch in Fig. 9(a)) are candidates for stitching (v_1 in Fig. 9(a)). Special care must be taken with self-loops in v_father in order for the process to finish and the stitch length to be meaningful. Also, in our implementation we have assumed that v_father did not have loops (except self-loops). In case v_father has loops we should make sure that the process finishes.

Starting with v_f, the method then attempts to find a reverse stitch that would potentially be longer. This is illustrated in Fig. 9(b), by examining vertices that cluster with v_father[v_f], such as v_2. The stitch can be extended in this way several times. However, since nothing prevents a vertex v and its v_father[v] from belonging to the same cluster, we must avoid stitching v_0 with itself.

All potential stitches are inserted in a priority queue, indexed with the length of the stitch. The method then empties the priority queue and applies the stitches in order of decreasing length until the vertex clustering is completely represented by stitches.

The next section discusses details of the variable-length method, that are important for a successful implementation. These details are not necessary, however, to understand the rest of this paper starting with Section 9.

8. Details of the variable-length method

8.1. Decoder order of connected components

The representation method must respect and use the decoder order of connected components of the manifold mesh. As mentioned in Section 1, independently of the number of vertices that cluster to a given vertex, geometry and properties for that vertex are encoded only once, specifically for the first vertex of the cluster that is decoded. Connectivity, stitches, geometry and properties are encoded and decoded on

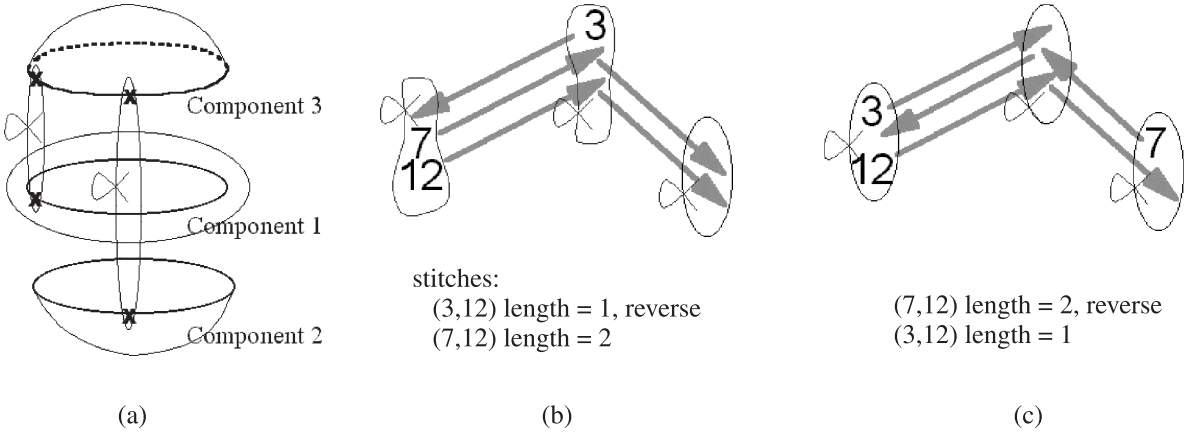


Fig. 10. Potential problems with variable-length stitches. (a) The clustering between components 1 and 2 is decoded only when component 3 is. (b) To successfully encode these two stitches we must substitute vertices 12 with 7 in the first one. (c) No possible re-combination using endpoints 3, 7 and 12 is possible.

a component-per-component basis to allow progressive decoding and visualization (see Fig. 4(c)). This implies that after decoding stitches corresponding to a given component, say component m, the complete clustering information (relevant portion of v_cluster) for component m as well as between component m and the previously decoded components $1, \ldots, m-1$ should be available. If this is not so, there is a mismatch between the geometry and properties that were encoded (too few) and those that the decoder is trying to decode, with potentially adverse consequences.

The stack-based method generates one command per vertex, for each cluster that is not trivial (cardinal larger than one), and will have no problem with this requirement. However, when applying the variable-length search for longest stitches on all components together, the optimum found by the method could be as in Fig. 10(a), where three components may be stitched together with two stitches, one involving components 1 and 3 and the second involving components 2 and 3.

Assuming that the total number of manifold components is c, our solution is to iterate on m, the component number in decoder order, and for m between 2 and c, perform a search for longest stitches on components $1, 2, \ldots, m$.

8.2. Decoder order of vertices

The longest stitch cannot always be performed, because of incompatibilities with the decoder order of vertices: a vertex can only be stitched to one other vertex of lower decoder order. The example in Fig. 10(b) illustrates this: the (12,3) and (12,7) stitches cannot be both encoded. However, the (12,3) stitch may be substituted with (7,3) which is an equally long stitch, and therefore listed in the priority queue. The case of Fig. 10(c) requires more work, because no possible re-combination is possible using endpoints 3, 7 and 12.

Since problems only involve vertices that start the stitch, it is possible to split the stitch in two stitches, one being one unit shorter and the other being of length zero. Both stitches are entered in the priority queue.

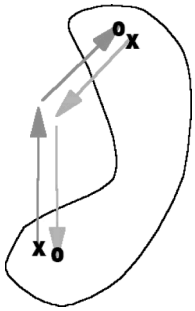


Fig. 11. Vertices marked with an "o" and an "x" may be stitched together, since this corresponds to the longest possible stitches. However, the complete clustering of four vertices (circled with the black curve) is not completely represented in this fashion.

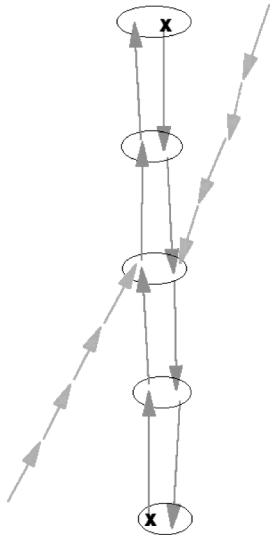


Fig. 12. A reverse stitch (between vertices marked with an "x") may be interrupted, because of a previous forward stitch, and vice versa.

For stitches of length zero, the incompatibility with the decoder order of vertices can always be resolved. In Fig. 10(b), for stitching 3 vertices, we can consider three stitching pairs, only one of which is being rejected. Since for stitches of length zero the direction of the stitch does not matter, all other stitching pairs are valid.

8.3. Generating enough stitches

The method generates the longest stitch starting at each vertex. It is possible that this may not provide enough stitches to encode all the clusters. This is illustrated in Fig. 11. In this case the method can finish encoding the clusters using zero-length stitches similarly to the stack-based method.

8.4. Competing forward and reverse stitches

Finally, because forward and reverse stitches "compete" with each other, the situation illustrated in Fig. 12 may occur: an isolated pair of vertices along a forward stitch may have been clustered by the operation of a reverse stitch that was performed earlier. To avoid redundancy, as the pair of vertices was already clustered, no subsequent stitch should incorporate them. Our method will detect this situation and split the stitch performed last in two shorter stitches.

Once a working combination of stitches is found, the last step is to translate them to stitching commands. This is the object of the next section which also specifies a bit-stream syntax.

9. Stitches encoding

To encode the stitching commands in a bit-stream, we propose the following syntax, that accommodates commands generated by both the stack-based and variable-length methods. To specify whether there are any stitches at all in a given component, a Boolean flag **has_stitches** is used. In addition to the PUSH, GET and POP commands, a vertex may be associated with a NONE command, as discussed above. In general, because a majority of vertices are expected to be non-singular, most of the commands should be NONE. Three bits called **stitching_command**, **pop_or_get** and **pop** are used for coding the commands NONE, PUSH, GET and POP as shown in Fig. 13.

A **stitch_length** unsigned integer is associated with a PUSH command. A **stack_index** unsigned integer is associated with GET and POP commands. In addition, GET and POP have the following parameters: **differential_length** is a signed integer representing a potential increment or decrement with respect to the length that was recorded with a previous PUSH command or updated with a previous GET and POP (using **differential_length**). **push_bit** is a bit indicating whether the current vertex should be pushed in the stack, [10] and **reverse_bit** indicates whether the stitch should be performed in a reverse fashion.

We now explain how to encode (translate) the stitches obtained in the previous sections in compliance with the syntax that we defined. Both encoder and decoder maintain an anchor_stack across manifold connected component for referring to vertices (potentially belonging to previous components). For the stack-based method, the process is straightforward: in addition to the commands NONE, PUSH, GET and POP encoded using the three bits **stitching_command**, **pop_or_get** and **pop**, a PUSH is associated with **stitch_length** $= 0$. GET and POP are associated with a **stack_index** that is easily computed from the anchor_stack.

For the variable-length method, the process can be better understood by examining Fig. 14. In Fig. 14(a) we show a pictorial representation of a stitch. A vertex is shown with an attached string of edges representing a stitch length, and a stitch_to arrow pointing to an anchor. Both vertex and anchor are represented in relation to the decoder order of (traversal of) vertices.

The stitch_to relationship defines a partition of the vertices associated with stitching commands. In Fig. 14(b) we isolate a component of this partition. For each such component, the method visits the vertices in decoder order (v_0, v_1, v_2, v_3 in Fig. 14(b)). For the first vertex, the command is a PUSH. Subsequent vertices are associated with a GET or POP depending on remaining stitch_to relationships; for vertices that are also anchors, a **push_bit** is set. Incremental lengths and **reverse_bit**s are also computed. Fig. 14(c) shows the commands associated with Fig. 14(b). For the example of Fig. 1 that we have used throughout this paper, the final five commands different from NONE are gathered in Table 2.

After the commands are in this form, the encoder operates in a manner completely symmetric to the decoder which is described in detail in Section 10, except that the encoder does not actually perform the stitches while the decoder does. Fig. 15 lists pseudo-code for the encoder.

[10] Since POP and GET have an associated **push_bit** there are fewer PUSH than POP commands (although this seems counter-intuitive). We have tried exchanging the variable length codes for PUSH and POP, but did not observe smaller bit-streams in practice; we attributed this to the arithmetic coder.

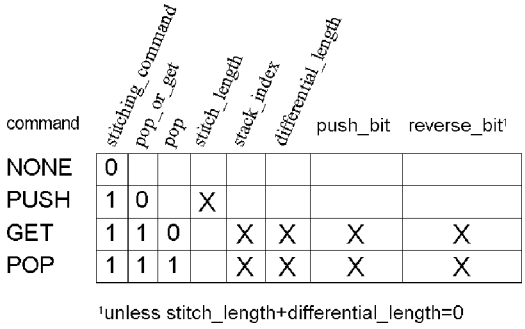


Fig. 13. Syntax for stitches. “X”s indicate variables associated with each command.

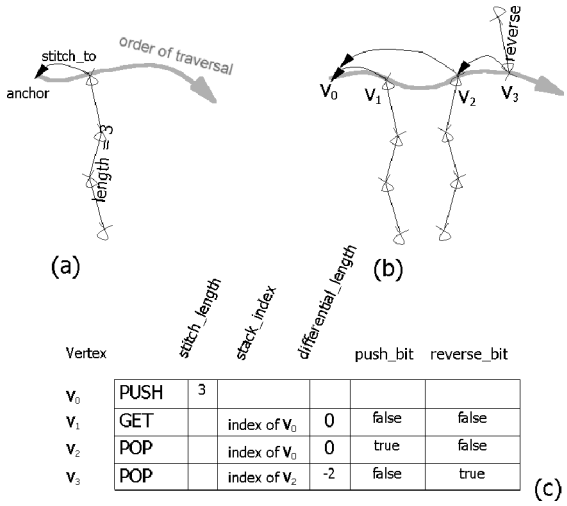


Fig. 14. Translating stitches to the bit-stream syntax.

Table 2
Five commands (different from NONE) encoding the complete clustering of Fig. 3. The stack-based encoding shown in Fig. 6 requires nine

Vertex	Command	**stitch _length**	**stack _index**	**differential _length**	**push _bit**	**reverse _bit**
0	PUSH	0				
1	PUSH	1				
5	POP		1	0	0	0
6	POP		0	0	1	0
0	POP		0	2	0	1

```
encoded(anchor_stack){
  if(has_stitches==true){
    encode has_stitches;
    for (i=nV0; i< nV1; i++){ //nV0 is the first vertex
    // of the current component, and nV1 –1 is the last vertex
      encode stitching_command;
        if(stitching_command){
          encode pop_or_get;
          if (pop_or_get){
            encode pop;
            encode stack_index,
            retrieve stitching_anchor from anchor_stack;
            if (pop){
              remove stitching_anchor from anchor_stack;
            } // end if
            encode incremental_length;
            if(incremental_length!=0){
              encode incremental_length_sign;
            } // end if
            decode push_bit;
            if(push_bit)
              push i to the back of anchor_stack
            retrieve stitch_length at stiching_anchor;
            total_length = stitch_length + incremental_length;
            if(total_length > 0)
              encode reverse_bit;
            save total_length at stitching_anchor;
          } // end if(pop_or_get)
          encode stitch_length;
          push i to the back of anchor_stack;
        } // end if(stitching_command)
    } // end for
}
```

Fig. 15. Pseudo-code for the stitches encoder.

10. Stitches decoding

The decoder reconstructs the v_cluster information that should be applied to vertices to reconstruct the polygonal mesh. The following pseudo-code shown in Fig. 16 summarizes the operation of the stitches decoder: if the Boolean **has_stitches** in the current connected component is true, then for each vertex of the current component in decoder order, a stitching command is decoded. If the Boolean value **stitching_command** is true, then the Boolean value **pop_or_get** is decoded; if the Boolean value **pop_or_get** is false, an unsigned integer is decoded, and associated to the current vertex i as an anchor (to stitch to). The current vertex i is then pushed to the back of the anchor_stack. If **pop_or_get** is true, then the Boolean value **pop** is decoded, followed with the unsigned integer value **stack_index**.

```
decode_stitches_for_a_connected_component (anchor_stack){
  if (has_stitches == true)
    for(i = nV0; i < nV1; i++){{ //nV0 is the first vertex
    // of the current component, and nV1 –1 is the last vertex
      decode stitching_command;
      if(stitching_command){
        decode pop_or_get;
        if(pop_or_get){
          decode pop;
          decode stack_index;
          retrieve anchor from anchor_stack;
          if(pop){
            remove stitching_anchor from anchor_stack;
          } // end if
          decode incremental_length;
          if (incremental_length!=0){
            decode incremental_length_sign;
          } // end if
          decode push_bit;
          if(push_bit)
            push i to the back of anchor_stack
          retrieve stitch_length at anchor;
          total_length = stitch_length + incremental_length;
          if(total_length > 0)
            decode reverse_bit;
          stitch i to anchor for length of total_length and in reverse if (reverse_bit);
        } // end if(pop_or_get)
        decode stitch_length;
        push i to the back to anchor_stack;
        save stitch_length at anchor i;
      } // end if(stitching_command)
  } // end for
}
```

Fig. 16. Pseudo-code for the stitches decoder.

Using **stack_index**, an anchor is retrieved from the anchor_stack. This is the anchor that the current vertex i will be stitched to. If the **pop** Boolean variable is true, then the anchor is removed from the anchor_stack. Then, an integer **differential_length** is decoded as an unsigned integer. If it is different from zero, its sign (Boolean **differential_length_sign**) is decoded, and is used to update the sign of **differential_length**. A **push_bit** Boolean value is decoded. If **push_bit** is true, the current vertex i is pushed to the back of the anchor_stack. An integer **stitch_length** associated with the anchor is retrieved. A total_length is computed by adding **stitch_length** and **differential_length**; if total_length is greater than zero, a **reverse_bit** Boolean value is decoded. Then the v_cluster array is updated by stitching the current vertex i to the stitching anchor with a length equal to total_length and potentially using a reverse stitch. The decoder uses the v_father array to perform this operation. To stitch the current vertex i to the stitching anchor with a length equal to total_length, starting from both i and the anchor at the same time,

we follow vertex paths starting with both i and the anchor by looking up the v_father entries total_length times, and for each corresponding entries (i,anchor), (v_father[i],v_father[anchor]), (v_father[v_father[i]], v_father[v_father[anchor]]), ... we record in the v_cluster array that the entry with the largest decoder order should be the same as the entry with the lowest decoder order. For instance, if (j > k), then v_cluster[j] = k else v_cluster[k] = j. v_cluster defines a graph that is a forest. Each time an entry in v_cluster is changed, we perform path compression on the forest by updating v_cluster such that each element refers directly to the root of the forest tree it belongs to.

If the stitch is a reverse stitch, then we first follow the v_father entries starting from the anchor for a length equal to total_length (from vertices 6 through 3 in Fig. 7), recording the intermediate vertices in a temporary array. We then follow the v_father entries starting from the vertex i and for each corresponding entry stored in the temporary array (from the last entry to the first entry), we update v_cluster as explained above.

11. Analysis

11.1. Correctness

We wish to determine if both the stack-based and variable-length methods can be used in combination with a manifold mesh encoding technique to encode any non-manifold mesh.

We first observe that a non-manifold mesh can always be converted to a set of manifold meshes by cutting through singular vertices and edges (including edges for which incident faces have an inconsistent orientation). The cut is performed by duplication of vertices. The inverse operation is to aggregate (cluster) the vertices that were duplicated. We may thus determine whether any clustering of some or all of the vertices of a mesh can be represented with either stack-based or variable-length method.

The stack-based method may clearly represent any clustering by virtue of the construction of Section 6 which we recall here briefly: a clustering of vertices is a partition of the set of vertices. Vertices may be globally enumerated. Inside each component of the partition, we may enumerate the vertices according to the global order. The first vertex is associated with a PUSH, the last vertex with a POP, and all intermediate vertices with a GET.

The variable-length method may also represent any clustering, for the simple reason that the variable-length method is a generalization of the stack-based method (if all stitches have a length of zero).

11.2. Computational complexity

We focus here on the computational complexity associated with the encoding and decoding of stitches; studying the computational complexity of manifold mesh encoding and decoding belongs to the relevant publications [1,9,12,14,22], and we only summarize the current analysis here: the authors of the above-referenced publication report a computational complexity that is linear in the number of mesh vertices and triangles for most methods, with for some methods a non-linear storage cost as pointed out in [17].

We now concentrate first on the computational complexity of encoding stitches, followed with the computational complexity of decoding.

The complexity is determined by the use of a stack-buffer. A subset of m vertices among the n vertices of the mesh are replicated, and there are a total of r replications. These r vertices are pushed and popped

inside a stack-buffer. We assume that at a given time, no more than k vertices are stored in the stack buffer. We necessarily have $k \leqslant m$. We need to determine the complexity of maintaining a stack-buffer allowing to index elements with indices between 0 and $k-1$. As vertices are added to the stack-buffer, we can use the depth in the stack as an index for vertices. However, when vertices are removed from the stack, we need to reassign unused indices without perturbing the indices of vertices that are still present in the stack (which need to be accessed directly using their original index). A queue may be used to track unused indices and reassign them (in any order). The cost of inserting or removing an element from the queue is constant. However, there is no guarantee to always assign the smallest index possible, which could have negative effects on the size of the encoding. This area is open for further investigation. We have thus established that stack-buffer operations may be performed in constant time.

However, indices between 0 and $k-1$ must be inserted in the bit-stream. Since we cannot assume any particular coherence between the indices, the worst-case cost of encoding these indices will be $O(\log k)$. Thus, the cost of processing all r vertices subjected to clustering will be bounded with $O(r \log k)$, and thus $O(r \log m)$ in the worst case.

In the case of the variable-length method, the process of determining the longest stitch starting at each of the m vertices has a worst-case complexity of $O(m^2)$, while the process of sorting stitches has no non-linear contribution (sorting integers can be done in linear time using Bucket Sort [4]).

In summary, encoding can be performed in $O(n + m \log m)$ time for the stack-based method, while the worst case for the variable-length method is bounded by $O(n + m^2)$.

In terms of decoding, except for the caveat formulated below, there is no difference in worst-case complexity between the two methods which is $O(n + m \log m)$, where again n is the number of vertices of the mesh and m is the number of vertices amongst n that are subjected to clustering.

In the case of the variable-length method, we note that a bad encoder could provoke situations as depicted in Fig. 8(b), where the clustering between vertices is redundantly encoded. We have assumed for our complexity estimate above that the encoder would not do this. However, the bit-stream syntax does not provide guarantees against this behavior.

11.3. Storage cost

The worst-case storage cost associated with the stitching is as follows. For the encoding, the stack-based method has a worst-case storage cost of m (maximum depth of the stack). The variable-length method has a worst-case storage cost of $O(mc)$, where c is the maximum number of vertices in a cluster as defined in Section 1. This cost corresponds to the storage of the set of candidate stitches.

For the decoding, both methods have a worst-case storage cost of m.

12. Experimental results

12.1. Test meshes

We report detailed data on a set of 14 meshes, and at the same time general statistics on a set of 303 meshes that were used for validation experiments during the MPEG-4 standardization process [11].

The 14 meshes are illustrated in Fig. 17. They range from having a few vertices (5) to about 65,000. The meshes range from having very few non-manifold vertices (2 out of 5056 or 0.04%) to a significant

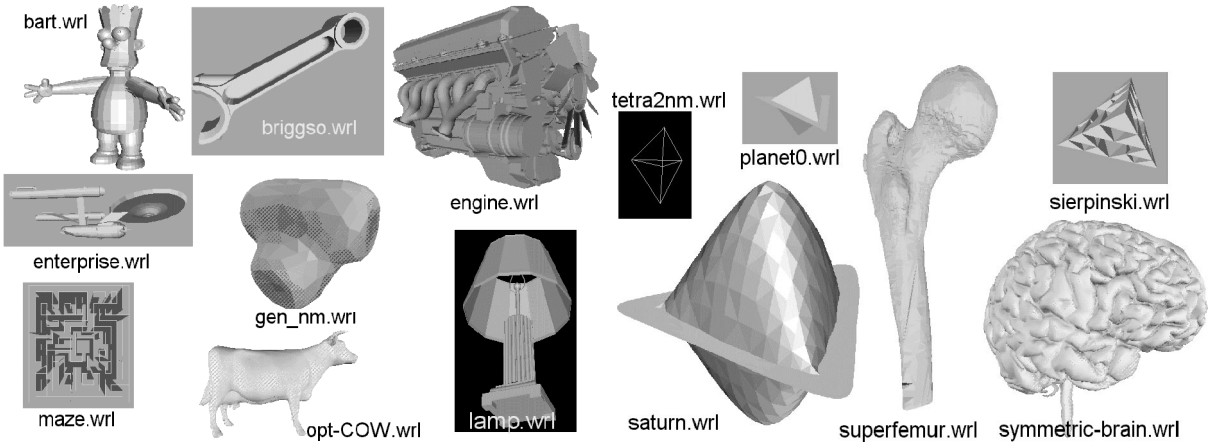


Fig. 17. Test meshes.

proportion of non-manifold vertices (up to 88 % for the Sierpinski.wrl model). One mesh was manifold and all the rest of the meshes were non-manifold. (The manifold mesh will be easily identified by the reader in Table 3.) One model (Gen_nm.wrl) had colors and normals per vertex. It was made non-manifold by adding triangles. The Engine model was originally manifold, and made non-manifold by applying a clustering operation as described in [15]. We synthesized the models Planet0.wrl, Saturn.wrl, Sierpinski.wrl, Tetra2nm.wrl. All other models were obtained from various sources and originally non-manifolds.

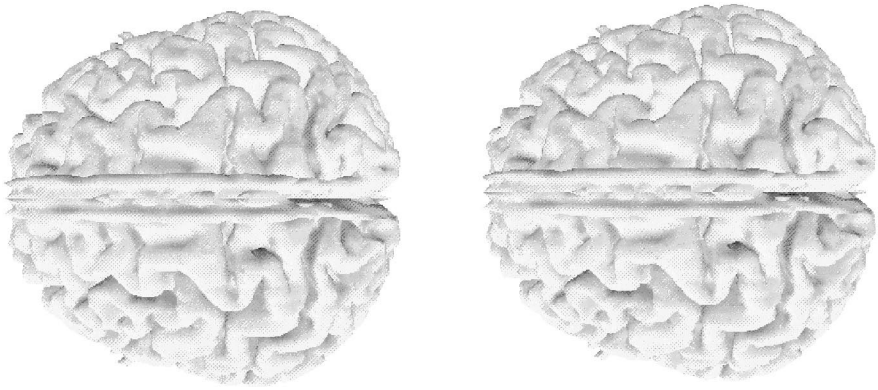
12.2. Test conditions

The following quantization parameters were used: geometry (vertex coordinates) was quantized to 10 bits per coordinate, colors to 6 bits per color, and normals to 10 bits per normal. The coordinate prediction was done using the "parallelogram prediction" [22], the color prediction was done along the triangle tree, and there was no normal prediction. Using 10 bits per coordinate, there are no noticeable differences between the original and decoded models in most cases. In the case of the Engine model, however, some quantization artifacts are visible when using 10 bits per coordinate, that disappear when using 16 bits per coordinate. We illustrate two of the larger test models before compression and after decompression in Figs. 18 and 19.

12.3. Test results

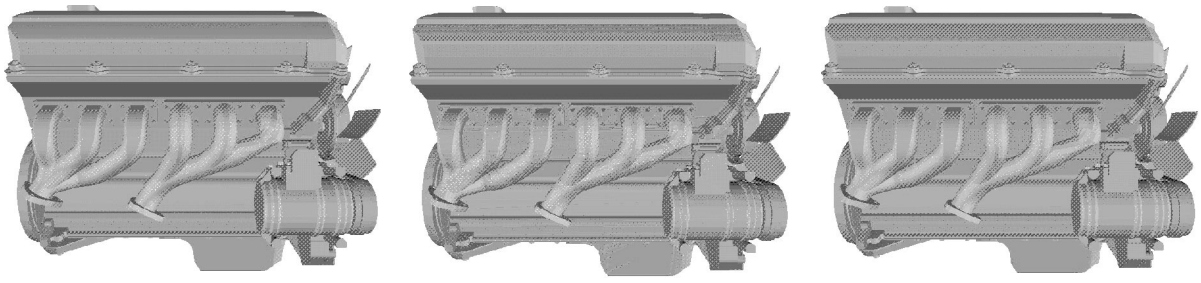
Table 3 provides compressed bit-stream sizes for the 14 meshes and compares the bit-stream sizes when meshes are encoded as non-manifolds or as manifolds (i.e., without the stitching information, and with redundent coordinates for the vertices that are repeated). There is an initial cost for each mesh on the order of 40 bytes or so, independently of the number of triangles and vertices.

In case of smooth meshes, the connectivity coding, prediction and arithmetic coding seem to divide by three or so the size of quantized vertices: for instance, starting with 10 bits per vertex of quantization, a typical bit-stream size would be on the order of 10 bits per vertex and 5 bits per triangle (assuming a



(a) 3,092 Kb (b) 74 Kb (17.2 bpv)

Fig. 18. (a) Symmetric-brain model before compression. (b) After decompression: starting 10 bits of quantization per vertex coordinate the complete compressed bit-stream uses 17.2 bits per vertex.



(a) 4,852 Kb (b) 139 Kb (17.8 bpv) (c) 267 Kb (33.6 bpv)

Fig. 19. (a) Engine model before compression. (b) After decompression (starting with 10 bits of quantization per vertex coordinate). Some artifacts may be seen at this level of quantization. (c) After decompression (16 bits). No visible artifacts remain.

manifold mesh without too many boundaries). In case of highly non-manifold or non-smooth meshes, starting with 10 bits per vertex of quantization, a typical bit-stream size would be on the order of 20 bits per vertex and 10 bits per triangle (smooth meshes seem to compress roughly twice as much).

The previous estimates apply to both manifold and non-manifold compression. Table 3 indicates that when compressing a non-manifold as a non-manifold (i.e., recovering the connectivity using stitches) the total bit-stream size can be reduced by up to 20% (21% for the “tetra 2nm. wrl” model). This is because when encoding stitches, vertices that will be stitched together are encoded only once (such vertices were duplicated during the non-manifold to manifold conversion process). The same applies to per-vertex properties.

Table 3
Compression results. "bpv" stands for "bits per vertex" and bpt for "bits per triangle"

Model	Uncompressed size	Number of vertices	Number of triangles	Compressed as non-manifold			Compressed as manifold	Non-manifold versus manifold	
	bytes			bytes	bpv	bpt	bytes	ratio	savings
Bart.wrl	392,030	5,056	9,000	7,243	11.46	6.43	8,105	0.89	11%
Briggso.wrl	130,297	1,584	3,160	4,080	20.61	10.32	4,129	0.98	2%
Engine.wrl	4,851,671	63,528	132,807	139,632	17.58	8.41	167,379	0.83	17%
Enterprise.wrl	859,388	12,580	12,609	28,224	17.95	17.91	29,553	0.95	5%
Gen_nm.wrl	49,360	410	820	2,566	50.06	25.03	2625	0.97	3%
Lamp.wrl	254,043	2,810	5,054	3,726	10.61	5.90	3954	0.94	6%
Maze.wrl	87,391	1,412	1,504	4,235	24.0	22.53	4855	0.87	13%
Opt-cow.wrl	204,420	3,078	5,804	7,006	18.02	9.66	7,006	1	0%
Planet0.wrl	1,656	8	12	82	82	54.6	96	0.85	15%
Saturn.wrl	61,155	770	1,536	1,998	20.75	10.40	2,197	0.91	9%
Sierpinski.wrl	4,702	34	64	193	45.64	24.12	252	0.76	4%
Superfemur.wrl	1,241,052	14,065	28,124	30,964	17.61	8.81	31,378	0.98	2%
Symmetric_brain.wrl	3,092,371	34,416	66,688	73,789	17.15	8.85	73,640	1.002	−0.2%
Tetra2nm.wrl	489	5	7	66	105.6	75.42	83	0.79	21%

The main results are gathered in Table 4. In Table 4, we first report the relative part (in %) of stitching information and connectivity. We observe that stitches can have a huge impact on the bitstream (up to 125% the size of connectivity). Thus, it is worthwhile to concentrate on an efficient encoding of stitches.

We then compare the relative efficiencies of the naive method that was described in Section 1.1 (encode a table of the repetitions such as Table 1), the stack-based method and the variable-length method. In order to have a fair comparison we measure the number of bits per replicated vertex that are used to encode the stitches. For the naive method, we use the formula $r \log_2 n$, where r is the number of repeated vertices, and n is the total number of vertices. Note that this formula underestimates the true formula $m \log_2 c + r \log_2 n$, and also does not model for the overhead of inserting data in a functional bitstream (that can be decoded incrementally, etc.): the formula thus reflects a theoretical best-case prediction. For the stack-based and variable-length methods, we report data obtained by producing bitstreams with and without the stitching data.

As expected, the variable-length method outperforms the stack-based method, which outperforms the naive method. In several cases, the variable-length method allows an order-of-magnitude improvement over the (theoretical) performance of the naive method.

We also report in this paper some statistics on a set of 303 models that were used for validation experiments during the MPEG-4 standardization process. Among 303 meshes, 162 were found to be non-manifold. We measured the average ratio of vertex replications (r/n) among the 303 models and found it to be equal to 0.39: on average, 39% of the vertices are repeated. While this seems quite high, we note that

Table 4
Relative importance of stitches and comparison between encoders. We report the relative part (in %) of stitches and the rest of the connectivity, and observe that stitches represent an important part. We also compare (in number of bits per replicated vertex) the naive method (using the formula $r \log_2 n$), the stack-based encoder and the variable-length encoder. Note that the formula underestimates the cost of the naive method, and disregards the overhead associated to putting data in a bit-stream

			Connectivity	Connectivity and stiches	Stitches/ connectivity	Naive	Stack	Variable-length
Model name	n	r		in bytes			bits/r	
bart	5058	4	294	303	3%	13	22	18
briggso	1631	90	448	509	14%	11	7.2	5.42
engine	83583	23173	17068	38432	125%	17	10.47	7.38
gen_nm	422	24	112	134	20%	9	7.33	7.33
lamp	3070	388	315	429	36%	12	6.02	2.35
maze	2028	1232	615	1121	82%	11	3.56	3.29
planet0	14	9	45	50	11%	4	4.44	4.44
saturn	866	144	195	209	7%	10	5.72	0.78
sierpinski	64	60	57	80	40%	7	3.07	3.07
superfemur	14558	863	3699	4609	25%	14	8.50	8.44
brain	34708	548	7582	8154	8%	16	9.08	8.35
tetra2nm	11	9	42	47	12%	4	5.33	4.44

r is in general significantly larger than m, the number of non-manifold vertices: for instance for the model of Fig. 2, $m = 3$, $r = 9$ and $r/n = 0.82$. This average is kept high by a few models consisting of a majority of non-manifold vertices. The median, computed on the 162 non-manifold meshes is $r/n = 0.147$: 15% of the vertices are repeated.

In Fig. 20, we compare the efficiency of the variable-length method and the naive method for representing the stitching information in the 162 above mentioned non-manifold meshes. Fig. 20 shows a scatter plot of the number of bits per replicated vertex as a function of the ratio r/n. For the naive method, we have used the formula $r \log_2 n$, which as discussed above represent a theoretical best-case estimate. For the variable-length method, we have plotted data obtained by producing bitstreams with and without the stitching information. This data indicates that the variable-length method outperforms significantly the (theoretical behavior of the) naive method.

Table 5 gathers overall encoding and decoding timings [11]. We observe a decoding speed of 10,000–13,000 vertices per second on a commonly available 233 MHz Pentium II laptop computer. For many meshes it has been reported that the number of triangles is about twice the number of vertices: this

[11] Which are, perhaps, more relevant for [11,19], the present methods representing only one module.

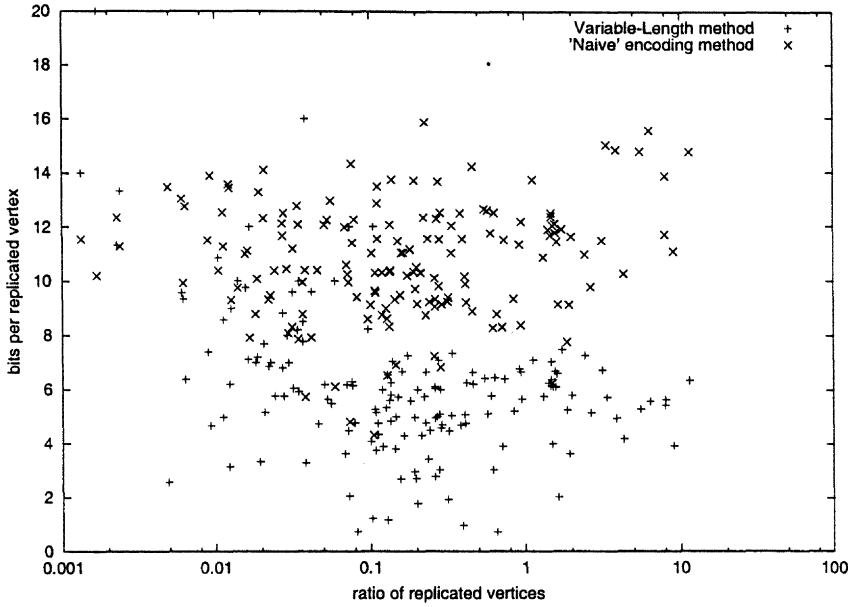


Fig. 20. Efficiency of the variable-length method versus the naive method for encoding the vertex replications: scatter plot obtained from 162 non-manifold models among 303 test models. The y-axis represents the number of encoded bits per replicated vertex, while the x-axis represents the ratio of vertex replications in log-scale. For the naive method, we have used the formula $r \log_2 n$ (theoretical best-case performance). For the variable-length method, we have plotted measured data. Based on this data, the variable-length method outperforms significantly the naive method.

is exact for a torus, and is approximate for many large meshes with a relatively simple topology. In this case we observe a decoding speed of 20,000–25,000 triangles per second. When considering non-manifold meshes the assumption that the number of triangles is about twice the number of vertices does not necessarily hold, depending on the number of singular and boundary vertices and edges of the model (for instance, consider the Enterprise.wrl model). This is why for non-manifold meshes, or meshes with a significant number of boundary vertices, when measuring computational complexity the number of vertices is probably a better measure of shape complexity than the number of triangles. The speed reported above is observed with most meshes, including meshes with one or several properties (such as gen_nm.wrl), with the exception of meshes with fewer than 50 vertices or so, which would not be significant for measuring per-triangle or per-vertex decompression speeds (because of various overheads).

While these results appear to be at first an order of magnitude slower than those reported in [9], we note that Gumhold and Strasser decode the connectivity only (which is only one functionality, and a small portion of compressed data) and observe their timings on a different computer (175 MHz SGI/02). Also, our decoder was not optimized so far (more on this in Section 13). Timings reported are independent of whether the mesh is a manifold mesh or not. There is thus no measured penalty in decoding time incurred by stitches.

Table 5
Encoding and decoding times in seconds measured on an IBM Thinkpad 600 233 MHz computer. The stack-based method was used. The encoding times include non-manifold to manifold conversion

Non-manifold	Encoding	Decoding	Vertices	Triangles
model	CPU time in seconds		Decoded/second	
Bart.wrl	0.64	0.38	13,300	23,700
Briggso.wrl	0.24	0.14	11,300	22,600
Engine.wrl	12.35	7.88	8,100	16,900
Enterprise.wrl	1.29	1.12	11,200	11,300
Gen_nm.wrl	0.10	0.04	10,300	20,500
Lamp.wrl	0.39	0.25	11,200	20,200
Maze.wrl	0.18	0.12	11,800	12,500
Cow.wrl	0.43	0.23	13,400	25,200
Planet0.wrl	0.02	0.02	400	600
Saturn.wrl	0.14	0.08	9,600	19,200
Sierpinski.wrl	0.03	0.02	1,700	3,200
Superfemur.wrl	2.12	1.36	10,300	20,700
Symmetric-brain.wrl	7.34	3.20	10,800	20,800
Tetra2nm.wrl	0.02	0.02	250	350

13. Summary and future work

We have described a method for compressing non-manifold polygonal meshes that combines an existing method for compressing a manifold mesh and new methods for encoding and decoding stitches. These latter methods comply with a new bit-stream syntax for stitches that we have defined.

While our work uses an extension of the Topological Surgery method for manifold compression [11], there are no major obstacles preventing the use of other methods such as [1,9,12,14,22].

13.1. Main results

We have demonstrated in this paper that compressing non-manifolds (while preserving their non-manifold connectivity) is highly desirable, and not very costly. Non-manifold models are frequent (more than half the models in our database). According to our experiments, non-manifold compression has no noticeable effect on decoding complexity. Furthermore, compared with encoding a non-manifold as a manifold, our method permits savings in the compressed bit-stream size (of up to 20%, and in average of 8.4%), because it avoids duplication of vertex coordinates and properties. This is in addition to achieving the functionality of compressing a non-manifold without perturbing the connectivity.

We have also demonstrated in this paper that encoding the stitching information efficiently is important. Our results indicate that the size of the stitching information may be comparable to the size of the connectivity. A naive method as discussed in Section 1.1 is not adequate for encoding the stitching information. Our methods can guarantee a worst case cost of $O(\log_2 m)$ bits per vertex replication, m being the number of non-manifold vertices, while for many replications, the cost is actually $\log l/l$ bits, where l is the length of the stitch (this is the amortized cost of encoding the length of the stitch).

We presented two different encoders: a simple encoder, and a more complex encoder that uses the full potential of the syntax. The results we reported indicate that the additional complexity of the variable-length encoder is justified. Other encoders may be designed in compliance with the syntax. One particularly interesting open question is: is there a provably good optimization strategy to minimize the number of bits for encoding stitches?

Perhaps more importantly, our methods hides completely the issues of mesh singularities to the users of the technology. These are arguably complex issues that creators and users of 3D content may not necessarily want to learn more about, in order to understand how the models would have to be freed of singularities (and thus altered) in order to be properly transmitted or stored in compressed form. The bitstream syntax and decoder described in this paper is part of the MPEG-4 standard on 3-D Mesh Coding. Using this technology, there will be no alteration of the connectivity, whether non-manifold or manifold.

13.2. Future work

Stitches allow more than connectivity-preserving non-manifold compression: merging components and performing all other topological transformations corresponding to a vertex clustering are possible. How to exploit these topological transformations using our stitching syntax (or other syntaxes) is another open area.

The software that was used to report results in this paper was by no means optimized. Optimization must thus be done in harmony with all the functionalities of compression (e.g., streamed and hierarchical transmission) and will be the subject of future work. The decoder may be optimized in the following ways (other optimizations are possible as well): (1) limiting modularity and function calls between modules, once the functionalities and syntax are frozen; (2) optimizing the arithmetic coding, which is a bottleneck of the decoding process (every single cycle in the arithmetic coder matters); (3) performing a detailed analysis of memory requirements, imposing restrictions on the size of mesh connected components, and limiting the number of cache misses in this way.

Acknowledgements

The anonymous reviewers provided excellent suggestions for improving our original draft. We also thank G. Zhuang, V. Pascucci and C. Bajaj for providing the Brain model, and A. Kalvin for providing the Femur model.

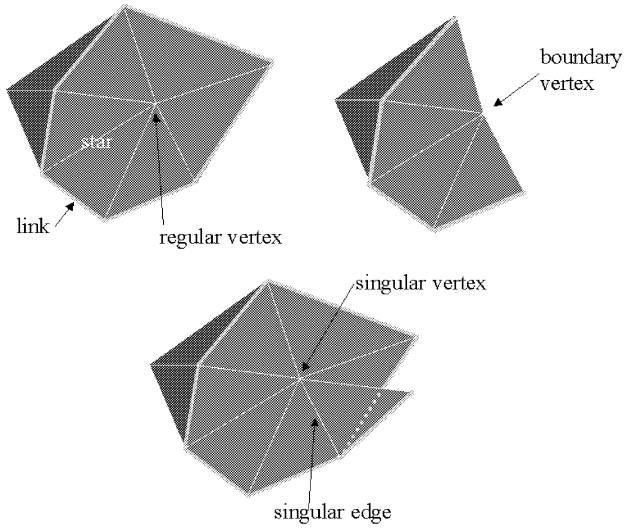


Fig. 21. Regular, boundary and singular (or non-manifold) vertices of a polygonal mesh.

Appendix A. Manifold and non-manifold polygonal meshes

For our purposes, a three-dimensional polygonal mesh comprises a set of vertices $\{\boldsymbol{v}_i\}$ and a set of faces $\{f_j\}$. Each vertex has coordinates in $\mathbb{R}^3$. Each face is specified with a tuple of at least three vertex indices. The face is said to be *incident* on such vertices. An *edge* is a pair of vertices listed consecutively in at least one face.

We call *connectivity* of a mesh, the set of ordered subsets of indices provided by the set of faces $\{f_j\}$, modulo circular permutation. We use the word *geometry* to mean the set of vertex coordinates $\{\boldsymbol{v}_i\}$.

We call the subset of faces of $\{f_j\}$ that share a vertex $\boldsymbol{v}$ the *star* of $\boldsymbol{v}$, noted $\boldsymbol{v}^\star$. The *link* of a vertex is a graph consisting of the edges of the star of $\boldsymbol{v}$ not incident to $\boldsymbol{v}$ (see Fig. 21). A *regular vertex* has a simply connected link; otherwise the vertex is a *singular vertex* or *non-manifold vertex*. We call an edge incident on one single face a *boundary edge*, an edge incident on exactly two faces a *regular edge*, and an edge incident on three or more faces a *singular edge*. A regular vertex incident to a boundary edge is called a *boundary vertex*. These cases are illustrated in Fig. 21. A mesh is a *manifold* if each vertex is a regular vertex; otherwise it is a *non-manifold*. Additional definitions (notably orientability) are provided, for instance, in [8].

References

[1] C. Bajaj, V. Pascucci, G. Zhuang, Single resolution compression of arbitrary triangular meshes with properties, in: Proceedings of Data Compression Conference, TICAM Report Number 99-05, 1999, pp. 247–256.
[2] Boeing research staff, personal communication, January 1999.
[3] M. Chow, Optimized geometry compression for real-time rendering, in: Visualization 97, Phoenix, AZ, October 1997, IEEE, pp. 415–421.
[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, McGraw-Hill, 1989.

[5] M. Deering, Geometry compression, in: Siggraph'95 Conference Proceedings, Los Angeles, August 1995, pp. 13–20.
[6] M. Denny, C. Sohler, Encoding and triangulation as a permutation of its point set, in: Proc. of the Ninth Canadian Conference on Computational Geometry, August 1997, pp. 39–43.
[7] A. Gueziec, F. Bossen, G. Taubin, C. Silva, Efficient compression of non-manifold polygonal meshes, in: Visualization'99, IEEE, San Francisco, CA, October 1999.
[8] A. Gueziec, G. Taubin, F. Lazarus, W.P. Horn, Converting sets of polygons to manifold surfaces by cutting and stitching, in: Visualization'98, IEEE, Raleigh, NC, October 1998, pp. 383–390.
[9] S. Gumhold, W. Strasser, Real time compression of triangle mesh connectivity, in: Siggraph'98 Conference Proceedings, Orlando, July 1998, pp. 133–140.
[10] H. Hoppe, Efficient implementation of progressive meshes, Computer and Graphics 22 (1) (1998) 27–36.
[11] ISO/IEC 14496-2 MPEG-4 Visual Committee Working Draft Version, SC29/WG11 document number W2688, Seoul, 2 April 1999.
[12] J. Li, C.C. Kuo, Progressive coding of 3D graphics models, Proceedings of the IEEE 96 (6) (1998) 1052–1063.
[13] J. Popovic, H. Hoppe, Progressive simplicial complexes, in: Siggraph'97 Conference Proceedings, Los Angeles, ACM, August 1997, pp. 217–224.
[14] J. Rossignac, Edgebreaker: Connectivity compression for triangle meshes, IEEE Trans. Visualization Comput. Graphics 5 (1) (1999) 47–61.
[15] J. Rossignac, P. Borrel, Multi-resolution 3d approximations for rendering, in: B. Falcidieno and T.L. Kunii (Eds.), Modeling in Computer Graphics, Springer, 1993, pp. 455–465.
[16] J. Rossignac, D. Cardoze, Matchmaker: manifold breps for non-manifold r-sets, in: SMA '99, Proceedings of the Fifth Symposium on Solid Modeling and Applications, Ann Arbor, MI, June 1999, ACM, pp. 31–41.
[17] J. Rossignac, A. Szymczak, Wrap&Zip decompression of the connectivity of triangle meshes compressed with Edgebreaker, Computational Geometry 14 (1–3) (1999) 119–135.
[18] M.J. Slattery, J.L. Mitchell, The Qx-coder, IBM J. Res. Develop. 42 (6) (1998) 767–784.
[19] G. Taubin, W.P. Horn, F. Lazarus, J. Rossignac, Geometry coding and VRML, Proceedings of the IEEE 86 (6) (1998) 1228–1243.
[20] G. Taubin, J. Rossignac, Geometry compression through topological surgery, ACM Trans. Graphics 17 (2) (1998) 84–115.
[21] The Virtual Reality Modeling Language Specification, VRML'97 Specification, June 1997, http://www.web3d.org/Specifications/VRML97.
[22] C. Touma, C. Gotsman, Triangle mesh compression, in: Proc. 24th Graphics Interface Conference, San Francisco, 1998, pp. 26–34.
[23] V. Abadjev, M. del Rosario, A. Lebedev, A. Migdal, V. Paskhaver, Metastream, in: Proc. VRML'99, Paderborn, Germany, February 1999, pp. 53–62.