Efficient Compression of Non-Manifold Polygonal Meshes *

André Guéziec † Frank Bossen ‡ Gabriel Taubin § Claudio Silva ¶

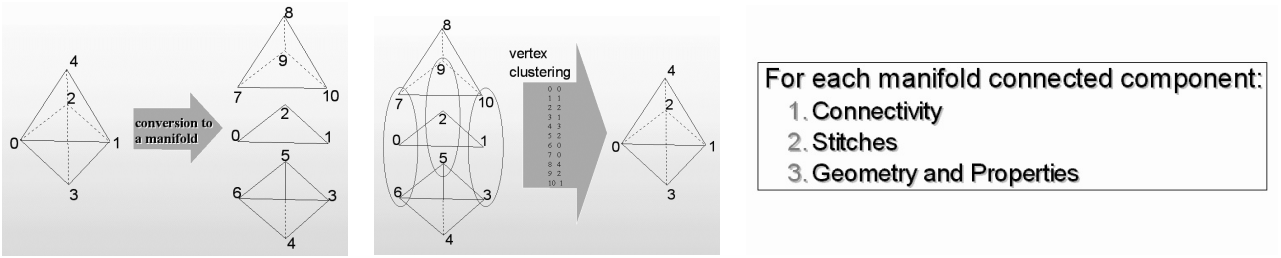


Figure 1: A: cutting a non-manifold mesh (two tetrahedra sharing a face). B: representing a non-manifold mesh as a manifold mesh together with a vertex clustering. C: Overall compressed syntax for a non-manifold mesh.

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 a conversion process, 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. By separating connectivity from geometry and properties, the method avoids encoding 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.

Key-words : 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 particular, polygons (especially, triangles) are required for generating three-dimensional renderings using available graphics architecture.

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. 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) [1]. In fact, on a database of 300 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 (157 exactly) were non-manifolds. As discussed in Section 2, most of the methods currently available for geometry compression require a manifold connectivity. Meshes can be converted, but then the original connectivity is lost, as discussed in our work [1]. At the time this paper is written, we are aware of only two publications treating connectivity-preserving non-manifold mesh compression [2, 3].

In this paper we describe a method for compressing non-manifold polygonal meshes and recovering their exact connectivity (and topology) after decompression. Our method compares in compression efficiency and speed with the most efficient manifold-mesh compression methods, thus extending [4, 5, 6, 7, 2], and even allows some savings by avoiding duplicate encodings of vertex coordinates and properties. method works by converting the original mesh to a set of manifold meshes, encoding the manifold meshes using an existing mesh compression technique, and clustering, or *stitching* together during the decompression process the vertices that were duplicated earlier to faithfully recover the original connectivity (see Fig. 1-A,B). To convert a non-manifold to a manifold by cutting (as little as possible), we are using the method described in [1] and briefly recalled in Section 3. However, there is significant flexibility in the strategies used for converting to manifold meshes and compressing them, and the present method does not require using specific ones. A *vertex clustering* [2] is recorded during the conversion process, such that when applied to the manifold meshes, the original non-manifold mesh is recovered.

Representation for Compression. The basic idea in this paper is to encode both the manifold meshes and vertex clustering as a substitute for the non-manifold mesh. While this idea is quite obvious, efficiently encoding and decoding a vertex clustering isn't, and will be the primary focus here. The mesh is compressed as indicated in Fig. 1-C. For each manifold connected component, the connectivity is encoded, followed with optional stitches, and geometry and properties. Stitches are used to recover the vertex clustering within the

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

†Multigen Paradigm, 550 S. Winchester Bvd., Suite 500, San Jose, CA 95128, gueziec@multigen.com

‡Signal Processing Lab, EPFL, 1015 Lausanne, Switzerland, frank.bossen@epfl.ch

§IBM T.J.Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, taubin@watson.ibm.com

¶AT&T Labs-Research, 180 Park Ave, P.O. Box 971, Florham Park, NJ 07932, csilva@research.att.com

[1] Specifically, a manifold polygonal surface is such that the neighborhood of every vertex can be continuously deformed to a disk (to a half disk at the boundary).

[2] I.e., means for specifying groups of vertices, such that each group should be clustered to form a single vertex.

current component and between vertices of the current 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).

As described in Section 5, the vertex clustering is decomposed in a series of variable-length *stitches*, that merge a given number of vertices along two directed paths of vertices. We propose to choose between a very simple and fast decomposition technique and a more advanced one. This latter decomposition presents several challenges that are described and overcome. The bit-stream syntax supports both possibilities, and it is not required that the encoder use the more advanced feature. In Sections 6 and 7, we give details on the encoding and decoding processes and describe a proposed bit-stream syntax for stitches.

2 Related Work

Connectivity-preserving non-manifold mesh compression algorithms were proposed by Popovic and Hoppe [2] and Bajaj *et al.* [3]. Hoppe's Progressive Meshes [8] 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 to 50 bits per vertex with arithmetic coding [8]. Utilizing more general vertex insertion strategies, this method was extended in [2] to represent arbitrary simplicial complexes, manifold or not, using about 50 bits per vertex (asymptotically the cost of this method is proportional to $n \log n$, n being the number of vertices). Our present method improves upon [2] 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 ring" mesh compression approach [3] partitions meshes in vertex and triangle layers that can represent a non-manifold mesh. Since vector quantization is used for compressing the geometry as opposed to scalar quantization in the present work, a direct comparison of the results is difficult. One advantage of the approach in this paper is that stitches may be used for encoding changes of topology (such as aggregating components) in addition to representing singularities.

Deering [9] 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 resending them. Deering's work fostered research on 3D mesh compression for other applications. Chow [10] extended [9] with efficient generalized-triangle-strip building strategies.

The Topological Surgery single-resolution mesh compression method [11, 12] 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 [12] also encodes the vertex coordinates (geometry) and all property bindings defined in VRML'97 [13].

Touma and Gotsman [4] traverse a triangular (or polygonal) mesh and remove one triangle at a time, recording vertex *valences*[3] as they go and recording triangles for which a boundary is split in two as a separate case.

Gumhold and Strasser [5] and Rossignac [6] concentrate on encoding the mesh connectivity. They use mesh traversal techniques similar to [4], 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 [14].

Li and Kuo [7]'s "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 [12]) and special commands to merge topologically distant polygons (to represent a general connectivity-not only a disk).

[3] Number of incident polygons.

3 Cutting a Non-manifold Mesh to Produce Manifold Meshes

We briefly recall here the method of Guéziec *et al.* [1] that we are using. For each edge of the polygonal mesh, it is determined whether the edge is singular (has three or more incident faces) or regular. Edges for which incident faces are inconsistently oriented are also considered to be singular for the purpose of this process. For each *singular* vertex of the polygonal mesh, the number of connected *fans* of polygons incident to it is determined[4]. 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. 1.

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

4 Compressing Manifold Meshes

The method described in this section extends the Topological Surgery method [12], and is explained in detail in [15]. In [12] 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 contructed 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 [12] geometry and properties are coded differentially with respect to a prediction. This prediction is obtained by a linear com-

[4] 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.

bination 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" [4][5] is a special case of this scheme, and is achieved through the appropriate selection of the weighting coefficients.

Coding efficency is further improved by the use of an efficient adaptive arithmetic coder [16]. 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 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

We introduce two methods, called Stack-Based and Variable-Length. The decomposition of the vertex clustering in stitches relies on the availability of two main elements: (1) a decoding order for the mesh vertices, and (2), for the variable-length method only, an unequivocal means of defining paths of vertices. We next suppose that such paths are recorded in an array called v_father, representing a function $\{1,\ldots,n\} \xrightarrow{\text{v\_father}} \{1,\ldots,n\}$, where n is the number of vertices.

All of the manifold mesh compression methods reviewed in Section 2 can provide an order in which vertices will be decoded as well as unambiguous paths of vertices from the decoded connectivity (the decoding order being one example). Fig. 2 shows v_father for the example of Fig. 1, obtained using the Topological Surgery method. The information consigned in v_father is implicit, 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). Both the stack-based and variable-length methods take as input a vertex clustering array, which for convenience we denote by v_cluster ($\{1,\ldots,n\} \xrightarrow{\text{v\_cluster}} \{1,\ldots,n\}$).

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 v_cluster$[1]$ = v_cluster$[256]$ = 1 than v_cluster$[1]$ = 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.

5.1 Stack-Based Method

We use a *stack-buffer* for stitches, similarly to Deering [9] and other manifold mesh compression modules (see [15]). In the decoding order, we push, get and pop in a stack-buffer[6] the vertices that cluster

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

[6] A "stack" would only support "push" and "pop" operations. We denote by "stack-buffer" a data structure that supports the "get" operation as well, i.e., direct indexing.

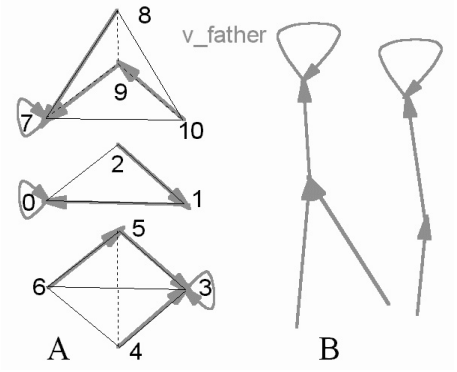


Figure 2: A: v_father for the example of Fig. 1. B: in the particular case of Topological Surgery [15], v_father is a forest that also admits self-loops. In the following, we will omit to draw self-loops.

together. Connected components (i.e., clusters) can be computed for the vertex clustering, such that two vertices belong to the same component if they cluster to the same vertex. We thus associate a *stitching command* for each vertex that belongs to a component whose size is larger than one. The command is either PUSH, or GET, or POP depending on the decoding order of the vertices in a given component. The vertex that is decoded first is associated with a PUSH; all subsequently decoded vertices are associated with a GET except the vertex decoded last, which is associated with a POP. For the example of Fig. 1 we illustrate the association of commands to vertices in Fig. 3.

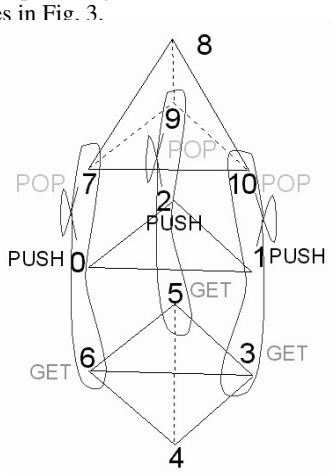


Figure 3: Stack-based method applied to the example of Fig.1.

5.2 Variable-Length Method

One drawback of the stack-based method is that it requires to send one stitching command (either PUSH, GET or POP) for each vertex that clusters to a singular vertex. Instead, by specifying an integer length, we could keep stitching vertex pairs when following the v_father relationship. This simple idea is illustrated in Fig. 4.

Using the same example of Fig 1, we illustrate in the Fig. 5 how variable length stitches can be used to represent the vertex clustering. A *stitch of length* l greater than zero is obtained by starting with two vertices and stitching vertices along two paths starting at the vertices and defined using the v_father graph, exactly $l+1$ times. For the example of Fig. 5, 3 stitches are applied to represent v_cluster: one (forward) stitch of length 1, one stitch of length zero, and one stitch of length 2 in the reverse direction. A stitch in the

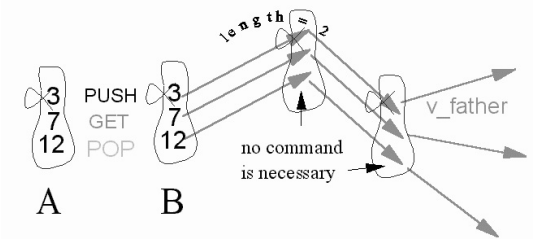


Figure 4: A: the stack-based method requires one command for each vertex that is clustered. B: with the variable-length method, the specification of a length can eliminate several commands.

reverse direction works similarly by starting with two vertices, following the path for the second vertex and storing all vertices along the path in a temporary structure, and stitching vertices along the first path together with the stored vertices visited in reverse order. In the remainder of this section, we explain how to discover such stitches from the knowledge of the v_cluster and v_father arrays.

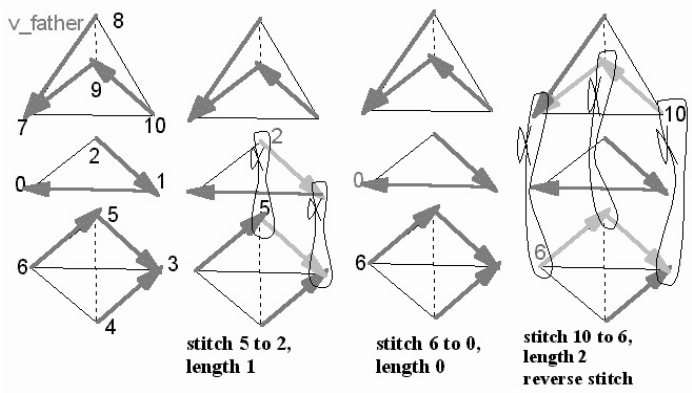


Figure 5: Three stitches of variable length and direction encode the vertex clustering of Fig. 1.

While our ultimate goal is to minimize the total encoding size for stitches (with manageable encoder and especially, decoder, complexities), a good working hypothesis (heuristic) 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. 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. 6-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 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. 6-A) are candidates for stitching (v_1 in Fig. 6-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. 6-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. This simple strategy must be extended to cope with the following issues (which are irrelevant for the stack-based method).

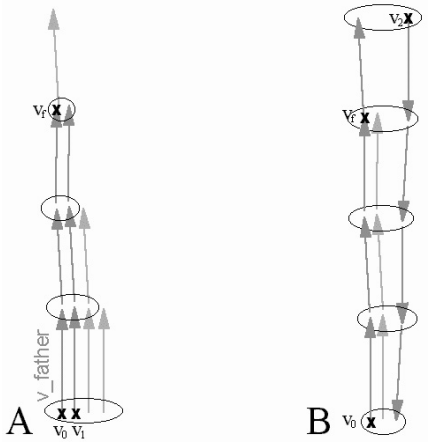


Figure 6: Computing the longest possible stitch starting at a vertex v_0. Ovals indicate clusters. A: forward stitch of length 3 with v_1. B: backward stitch of length 4 with v_2.

1. The representation method must respect and use the decoder order of connected components of the manifold mesh. As mentioned in the Introduction, 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 a component-per-component basis (see Fig. 1-C) to allow progressive decoding and visualization. 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. 7-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 $1, 2, \ldots, m$.

2. 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. 7-B illustrates this: the (12,3) and (12,7) stitches cannot be both encoded.

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. For stitches of length zero, the incompatibility with the decoder order of vertices can always be resolved. In Fig. 7-B, for stitching 3 vertices, we can consider three stitching pairs, only one of which being rejected. Since for stitches of length zero the direction of the stitch does not matter, all other stitching pairs are valid.

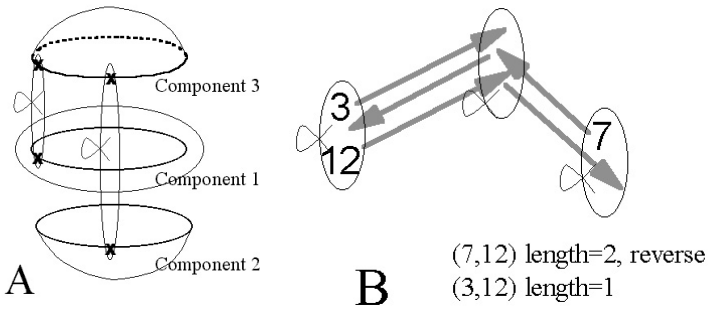


Figure 7: Potential problems with variable-length stitches. A: the clustering between Components 1 and 2 is decoded only when Component 3 is. B: These two stitches cannot be both encoded, because Vertex 12 can only be stitched to *one* vertex of lower decoding order (either 3 or 7 but not both.)

3. 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. In this case the method can finish encoding the clusters using zero-length stitches similarly to the stack-based method.

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.

6 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, in case it is sole representative of its cluster (e.g. does not correspond to a singular vertex in the non-manifold mesh), or in case the information on how to cluster it was already taken care of (variable-length method only). 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. 8.

command	stitching_command	pop_or_get	pop	stitch_length	stack_index	differential_length	push_bit	reverse_bit[1]
NONE	0							
PUSH	1	0		X				
GET	1	1	0		X	X	X	X
POP	1	1	1		X	X	X	X

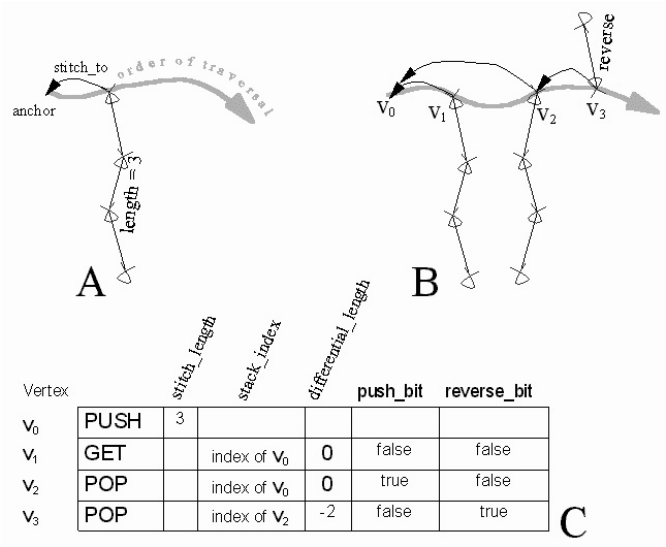
[1]unless stitch_length+differential_length=0

Figure 8: Syntax for Stitches. "X"s indicate variables associated with each command

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, [7] 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 section in compliance with the syntax 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. 9. In Fig. 9-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.



Vertex	stitch_length	stack_index	differential_length	push_bit	reverse_bit
v_0	PUSH	3				
v_1	GET		index of v_0	0	false	false
v_2	POP		index of v_0	0	true	false
v_3	POP		index of v_2	-2	false	true

C

Figure 9: Translating stitches to the bit-stream syntax.

The stitch_to relationship defines a partition of the vertices associated with stitching commands. In Fig. 9-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. 9-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. 9-C shows the commands associated with Fig. 9-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 1.

After the commands are in this form, the encoder operates in a manner completely symmetric to the decoder which is described in detail in the next section, except that the encoder does not actually perform the stitches while the decoder does.

[7] 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.

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

Table 1: Five commands (different from NONE) encoding the complete clustering of Fig. 1. The stack-based encoding shown in Fig. 3 requires nine.

7 Stitches Decoding

The decoder reconstructs the v_cluster that should be applied to vertices to reconstruct the polygonal mesh. The following pseudo-code shown in Fig. 10 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 of anchor_stack;
     save stitch_length at anchor i;
   } // end if (stitching_command)
  } // end for
}
```

Figure 10: 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 to total_length (from Vertices 6 through 3 in Fig. 5), 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.

8 Experimental Results

Test Meshes: 14 meshes illustrated in Fig. 11 (of the color page) were considered, ranging 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 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 2.) 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. We synthesized the models Planet0.wrl, Saturn.wrl, Sierpinski.wrl, Tetra2nm.wrl. All other models were obtained from various sources and originally non-manifolds.

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" [4], the color prediction was be done along the triangle tree (see Section 4), and there was no normal prediction. Using 10 bits per coordinate, there was hardly a noticeable difference between the original and decoded models. For completeness, we illustrate the Symmetric-brain test model before compression and after decompression in Fig. 12 (of the color page).

Test Results: This paper focuses about encoding and decoding stitches, which is a small portion of the geometry compression process. Our goal in this section is to determine how stitches affect the entire compression process. We will thus provide estimates of compression ratios and decoding timings that apply to the entire process, keeping in mind that the bulk of the compression process is described in other publications [12, 15]. The following estimates (obtained using the 14 meshes) may have to be revised as more statistical data becomes available, or as more efficient encoders and decoders are implemented.

Table 2 provides compressed bit-stream sizes for the 14 meshes and compares the bit-stream sizes when meshes are encoded as non-

Model	Uncompressed Size bytes	Number of Vertices	Number of Triangles	Compressed as Non-Manifold			Compressed as Manifold	Non-manifold vs Manifold	
				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%

Table 2: Compression results. "bpv" stands for "bits per vertex" and bpt for "bits per triangle"

Non-manifold Model	Stack-Based Encoder bit-stream size in bytes	Variable-Length Encoder bit-stream size in bytes	size ratio
Bart.wrl	7,245	7,243	1.0003
Briggso.wrl	4,100	4,080	1.005
engine.wrl	148,601	139,632	1.064
Gen_nm.wrl	2,566	2,566	1
Lamp.wrl	3,904	3,726	1.05
Maze.wrl	4,278	4,235	1.01
Planet0.wrl	82	82	1
Saturn.wrl	2,087	1,998	1.045
Sierpinski.wrl	193	193	1
Superfemur.wrl	30,971	30,964	1.0002
Symmetricbrain.wrl	73,839	73,789	1.0007
Tetra2nm.wrl	67	66	1.015

Table 3: Comparing the efficiency of the variable-length encoder vs. the stack-based encoder. The total bit-stream sizes are in bytes. 5% percent of the total bit-stream size represents a significant proportion of the connectivity (perhaps all of it) and is thus very significant for stitches.

Non-manifold Model	Encoding CPU Time in seconds	Decoding CPU Time in seconds	Vertices Decoded/second	Triangles 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

Table 4: Encoding and decoding times in seconds measured on an IBM Thinkpad 600 233MHz computer. The stack-based method was used. The times include non-manifold to manifold conversion.

manifolds or as manifolds (i.e., without the stitching information). There is an initial cost for each mesh on the order of 40 bytes or so, independently of the number of triangles and vertices. Although we do not provide specific results on the connectivity encoding in this paper, from data that we collected independently of the present study involving non-manifolds, we expect the connectivity to generally consume significantly fewer bits than coordinates and properties once compressed and arithmetic-coded (a few bits per triangle at most: from 0.1 bits to 3 bits per triangle).

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 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 compress roughly twice as much).

The previous estimates apply to both manifold and non-manifold compression. Table 2 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 tetra2nm.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 compares the efficiencies of the stack-based encoder and variable-length encoder by measuring total bit-stream sizes. The observed bitstream sizes decrease using the variable-length encoder, in three cases by about 5%. 5% of the total bit-stream size represents a significant proportion of the connectivity (perhaps all of it), while the stitches would represent a small portion of the connectivity (which includes vertex graph, triangle trees, etc.). Thus the savings of the variable-length encoder are very significant. These bits would be better used for a more accurate encoding of the geometry.

Table 4 gathers overall encoding and decoding timings using the stack-based method[8]. We observe a decoding speed of 10,000 to 13,000 vertices per second on a commonly available 233MHz Pentium II laptop computer. For many meshes it has been reported that the number of triangles is about twice the number of vertices: this is exact for a torus, and is approximate for many large meshes with

[8]Which are perhaps more relevant for [12, 15], the present methods representing only one module.

a relatively simple topology. In this case we observe a decoding speed of 20,000 to 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 above estimates apply to 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 [5], 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 (175MHz SGI/02). Also, our decoder was not optimized so far (more on this in Section 9). 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.

9 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 bitstream syntax for stitches that we have defined.

While our work uses an extension of the Topological Surgery method for manifold compression [15], there are no major obstacles preventing the use of other methods such as [4, 5, 6, 7, 3].

We reported results showing that 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 bitstream 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.

In terms of encoding, we presented two different encoders: a simple entry-level 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 second encoder is justified in several cases. 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?

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 such topological transformations using our stitching syntax (or other syntaxes) is another interesting avenue for future developments.

The technology described in this paper is part of the MPEG-4 standard on 3-D Mesh Coding. It hides completely the issues of mesh singularities to the user. These are arguably complex issues that creators and users of 3-D content may not necessarily want to learn about. Using the methods described in this paper, there will be no alteration of the original connectivity, whether non-manifold or manifold.

Decoder Optimization The software that was used to report results in this paper was by no means optimized. This is because non-manifold compression is only one of the functionalities of geometry compression, incremental (i.e., streamed) and hierarchical transmission being examples of other functionalities. Optimization must thus be done in harmony with all the functionalities 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.

Acknowledgments We thank G. Zhuang, V. Pascucci and C. Bajaj for providing the Brain model, and A. Kalvin for providing the Femur model.

References

[1] A. Gueziec, G. Taubin, F. Lazarus, and W.P. Horn. Converting sets of polygons to manifold surfaces by cutting and stitching. In *Visualization'98*, pages 383–390, Raleigh, NC., October 1998. IEEE.

[2] J. Popovic and H. Hoppe. Progressive simplicial complexes. In *Siggraph'97 Conference Proceedings*, pages 217–224, Los Angeles, August 1997. ACM.

[3] C. Bajaj, V. Pascucci, and G. Zhuang. Single resolution compression of arbitrary triangular meshes with properties. In *Proceedings of Data Compression Conference*, pages 247–256, 1999. TICAM Report Number 99-05.

[4] C. Touma and C. Gotsman. Triangle mesh compression. In *Proc. 24th Graphics Interface Conference*, pages 26–34, San Francisco, 1998.

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

[6] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, 1999.

[7] J. Li and C.C. Kuo. Progressive coding of 3D graphics models. *Proceedings of the IEEE*, 96(6):1052–1063, June 1998.

[8] H. Hoppe. Efficient implementation of progressive meshes. *Computer and Graphics*, 22(1):27–36, 1998.

[9] M. Deering. Geometry compression. In *Siggraph'95 Conference Proceedings*, pages 13–20, Los Angeles, August 1995.

[10] M. Chow. Optimized geometry compression for real-time rendering. In *Visualization 97*, pages 415–421, Phoenix, AZ., oct 1997. IEEE.

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

[12] G. Taubin, W.P. Horn, F. Lazarus, and J. Rossignac. Geometry coding and VRML. *Proceedings of the IEEE*, 86(6):1228–1243, Jun 1998.

[13] *The Virtual Reality Modeling Language Specification, VRML'97 Specification*, June 1997. http://www.web3d.org/Specifications/VRML97.

[14] M. Denny and C. Sohler. Encoding and triangulation as a permutation of its point set. In *Proc. of the Ninth Canadian Conference on Computational Geometry*, pages 39–43, August 1997.

[15] *ISO/IEC 14496-2 MPEG-4 Visual Committee Working Draft Version, SC29/WG11 document number W2688, Seoul, April 2nd, 1999.*

[16] M.J. Slattery and J.L. Mitchell. The Qx-coder. *IBM J. Res. and Dev.*, 42(6):767–784, 1998.

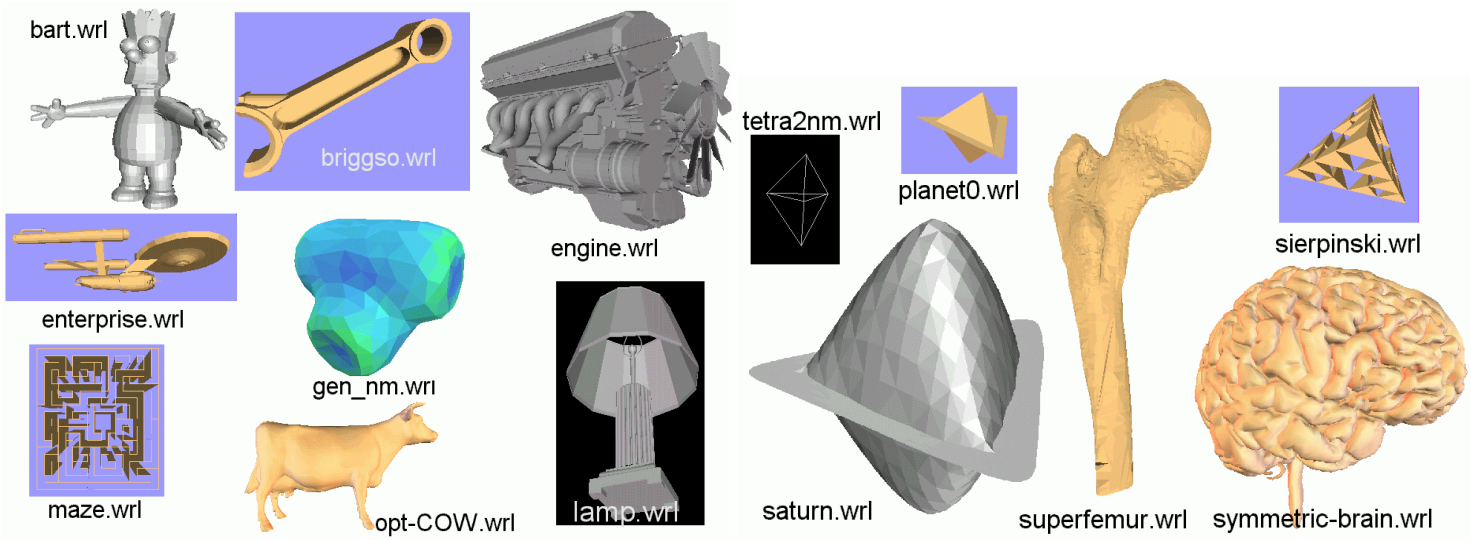


Figure 11: Test meshes.

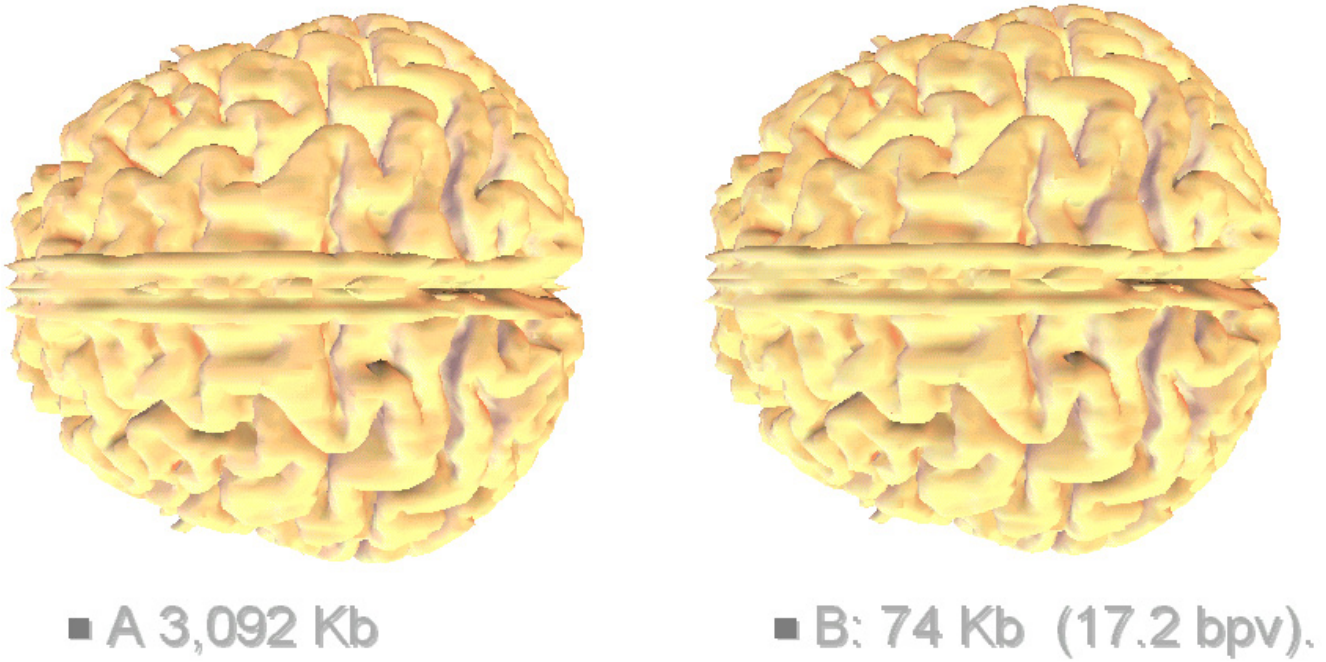


Figure 12: A: Symmetric-brain model before compression. B. after decompression: starting with 10 bits of quantization per vertex coordinate the complete compressed bitstream uses 17.2 bits per vertex.