Real Time Compression of Triangle Mesh Connectivity

Stefan Gumhold, Wolfgang Straßer*

WSI/GRIS University of Tübingen

Abstract

In this paper we introduce a new compressed representation for the connectivity of a triangle mesh. We present local compression and decompression algorithms which are fast enough for real time applications. The achieved space compression rates keep pace with the best rates reported for any known global compression algorithm. These nice properties have great benefits for several important applications. Naturally, the technique can be used to compress triangle meshes without significant delay before they are stored on external devices or transmitted over a network. The presented decompression algorithm is very simple allowing a possible hardware realization of the decompression algorithm which could significantly increase the rendering speed of pipelined graphics hardware.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture; I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

Keywords: Mesh Compression, Algorithms, 3D Graphics Hardware, Graphics

1 Introduction and Related Work

The ability to handle very large geometric data sets becomes more and more important. Powerful compression techniques are mandatory to solve this task. The compression approach presented in this paper has several advantages. The compression and decompression algorithms are fast and simple. The achieved space compression ratios are among the best known results and the algorithms act locally such that only a fraction of the vertices must be accessible. None of the available techniques combines these properties. All lack a very fast compression algorithm and therefore the compressed representation must be pre-computed before rendering. No speed up through compression could be achieved so far in the visualization of meshes with changing connectivity. The related work concentrates either on fast rendering or on maximum compression and therefore the following discussion is divided into two sections.

1.1 Compression for Fast Rendering

In this section we discuss representations of triangle meshes that are used for transmission to graphics hardware. 3D-hardware support is primarily based on the rendering of triangles. Each triangle is specified by its three vertices, where each vertex contains three coordinates, possibly the surface normal, material attributes and/or texture coordinates. The coordinates and normals are specified with floating point values, such that a vertex may contain data of up to 36 bytes[1]. Thus the transmission of a vertex is expensive and the simple approach of specifying each triangle by the data of its three vertices is wasteful as for an average triangle mesh each vertex must be transmitted six times.

The introduction of triangle strips helped to save unnecessary transmission of vertices. Two successive triangles in a triangle strip join an edge. Therefore, from the second triangle on, the vertices of the previous triangle can be combined with only one new vertex to form the next triangle. As with each triangle at least one vertex is transmitted and as an average triangle mesh has twice as many triangles as vertices, the maximal gain is that each vertex has to be transmitted only about two times. Two kinds of triangle strips are commonly used – the *sequential* and the *generalized triangle strips*. In generalized triangle strips an additional bit is sent with each vertex to specify to which of the free edges of the previous triangle the new vertex is attached. *Sequential strips* even drop this bit and impose that the triangles are attached alternating. OpenGL [7] which evolved to the commonly used standard for graphics libraries allowed [5] generalized triangle strips in earlier versions, but the current version is restricted to sequential strips. Therefore, the demands on the stripping algorithms increased. None of the existing algorithms reaches the optimum that each vertex is transmitted only twice. The algorithm of Evans et al. [4] produces strips such that each vertex is transmitted about 2.5 times and this is currently the best algorithm.

Arkin et al. [1] examined the problem of testing whether a triangulation can be covered with one triangle strip. For generalized triangle strips this problem is NP-complete, but for sequential strips there exists a simple linear time algorithm. But no results or algorithms were given to cover a mesh with several strips.

To break the limit of sending each vertex at least twice Deering [3] suggests the use of an on-board vertex buffer of sixteen vertices. With this approach, which he calls *generalized mesh* theoretically only six percent of the vertices have to be transmitted twice. But up to now no corresponding algorithms have been presented. Bar-Yehuda et al. [2] examined different sized vertex buffers. They prove that a triangle mesh with n vertices can be optimally rendered, i.e. each vertex is transmitted only once, if a buffer for $12.72\sqrt{n}$ vertices is provided. They also show that this upper bound is tight and no algorithm can work with less than $1.649\sqrt{n}$ buffered vertices. In their estimation they neglect the time consumed by the referencing of buffered vertices, which makes it impossible to determine the suitability of the approach for connectivity compression. Again the algorithms to compute the rendering sequences are not fast enough for on-line generation.

Our connectivity compression technique also utilizes a vertex buffer where each vertex has to be transmitted only once. As our technique is hard to analyze theoretically, we can only give experimental results of the size. These are all less than $12.72\sqrt{n}$. By defining a fixed traverse order our approach minimizes the number

*Email: {sgumhold/strasser}@gris.uni-tuebingen.de

[1] where we assumed four bytes per floating point value and one byte per color component

of indices needed to reference vertices in the buffer, which results in an additional speed up for rendering. If these indices are Huffman-encoded, in the average only two bits per triangle are needed for references.

1.2 Maximum Mesh Compression

The work on fast rendering explained in section 1.1 can also be used for the compression of triangle mesh connectivity. Instead of retransmitting a vertex, a reference is inserted into a compressed representation. If a vertex from the buffer is referenced its index within the buffer enters the compressed representation. In the triangle strips of Evans et al. [4] each vertex appears about 2.5 times. The vertices can be rearranged into the order they appear the first time and only the indices of $1.5n$ vertices need to be inserted in the compressed representation. One additional bit per triangle is needed to specify whether the next vertex is used the first time or the index of an already used vertex follows. This sums up to about $1 + 0.75 \cdot \lceil \log_2 n \rceil$ bits per triangle. The disadvantage of this approach is that the storage needs grow with the size of the triangle mesh. The measurements of Deering in [3] show that the generalized mesh approach theoretically consumes between eight and eleven bits per triangle if an optimal stripper is available.

The work of Bar-Yehuda et al. [2] cannot be compared to our work as no appropriate measurements are available.

Taubin et al. [8] propose a very advanced global compression technique for the connectivity of triangle meshes. The method is based on a similar optimization problem as for sequential triangle strips and the authors guess that it is NP-complete. Their approximation allows compression to only two bits per triangle and there exist triangle meshes which consume only one bit per triangle. The decompression splits into several processing steps over the complete mesh, which makes the approach unsuitable to speed up hardware driven rendering. Their compression and decompression algorithms are more complex than ours and although the asymptotic running time should be the same we strongly believe that a comparable optimized implementation of our algorithms is several times faster than the algorithms proposed by Taubin. Our compression technique yields nearly equivalent compression of the mesh connectivity.

So far we described only lossless compression techniques. For applications which allow lossy compression also the vertex data can be compressed and the connectivity can be simplified. Deering [3] uses the proximity of the vertices independently of the connectivity. Taubin et al. [8] propose to predict the coordinates v_n of the vertex, which is to be compressed next, from the preceding K vertices $v_{n-1}, \ldots, v_{n-K}$ and only encode the difference $\epsilon(v_n)$ to the predicted position

$$\epsilon(v_n) = v_n - P(\lambda, v_{n-1}, \ldots, v_{n-K}),$$

where λ specifies the parameters for the predictor function P, which was implemented as the linear filter $\sum_{i=1}^{K} \lambda_i v_i$ and the parameters where chosen to minimize the square sum of all $\epsilon(v_n)$. In both approaches of vertex data compression the positions are additionally entropy encoded after the delta compression. Similar techniques are used to compress the normals and the material data. Our approach of connectivity compression can be used with both of these geometry compression techniques. The result is a significant speed up in both cases and the improvement of the compression ratios of Deering's approach[2].

[2]The compression ratios for the models (triceratops, galleon, viper, 57chevy, insect) measured in Deering's paper would increase from (5.8X, 8.2X, 9.2X, 9.2X, 7.2X) to (7.4X, 11.2X, 11.6X, 12.0X, 11.4X)

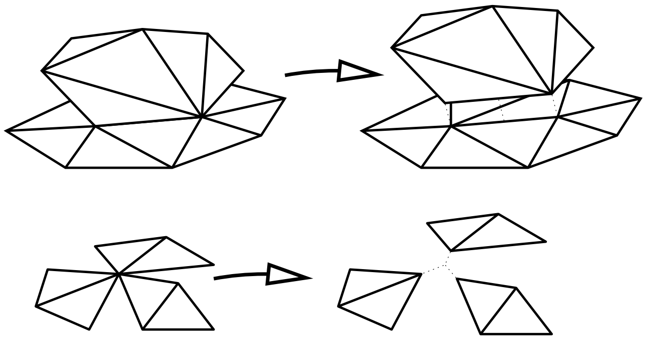

Figure 1: Non manifold vertices must be duplicated in order to make their neighborhood 2-manifold with border.

2 Compression and Decompression

Let us introduce the ideas of compression and decompression by comparison with generalized triangle strips. This approach utilizes a vertex buffer of only three vertices but in turn has to transmit each vertex twice. Thus the first idea is to simply increase the size of the vertex buffer, such that all the undecided vertices can be stored. The undecided vertices are those, which have not finally been incorporated into the so far decompressed triangle mesh, i.e. for these vertices exist adjacent triangles which will be transmitted later.

The increased vertex buffer is not very useful, if still indices of the vertices must be transmitted to localize them within the buffer. The transmission of most indices can be avoided by fixing the order in which the edges formed by the vertices in the vertex buffer are traversed. Therefore, the traverse order need not be encoded. The rules to fix the traverse order must be chosen carefully as they constitute most degrees of freedom for optimization. In turn the compression algorithm becomes nearly as fast as the decompressed algorithm. In the case of generalized triangle strips the traverse order is not fixed. Each of the additional bits encodes which of the two free edges of the previous triangle the next triangle is attached to.

To allow the encoding of an arbitrarily connected and oriented triangle mesh in one run, several basic building operations must be encodable. In the case of triangle strips there is only one operation – the attachment of a new triangle to an existing edge. This has the advantage that the type of operation need not be encoded. In our approach all of the building operations also introduce one new triangle, but the new triangle can additionally be formed exclusively by buffered vertices.

Section 2.1 introduces the rules which fix the order in which the triangle mesh is traversed. Then the different building operations are discussed in section 2.2. The building operations are Huffman encoded in a variable length bit stream to achieve the best compression of the connectivity. Section 2.3 explains how to combine the bit stream with the vertex data and additional data.

In what follows we assume that the triangle meshes consist of several connected components, all orientable and locally 2-manifold with borders. Thus the neighborhood of each vertex can be continuously mapped to a plane or to a half-plane if the vertex belongs to the border of a component. Our approach is extended to non orientable 2-manifold triangle meshes in section 5.1. Non manifold models must be cut into manifold models by duplication of non manifold vertices as indicated in figure 1.

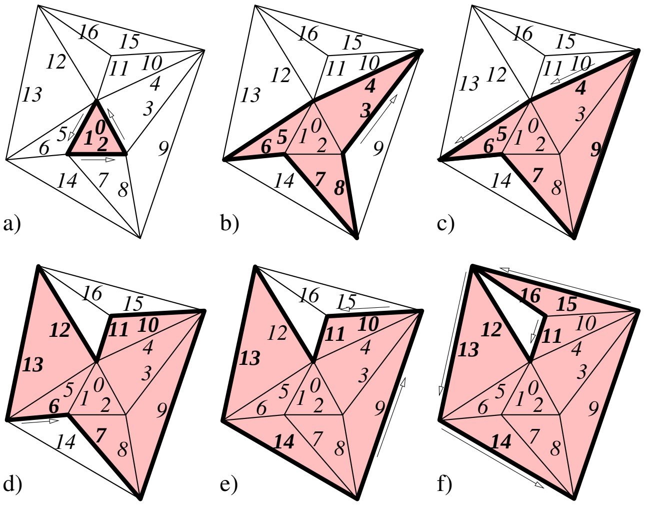

Figure 2: The shown sample triangle mesh is traversed in a breadth-first order.

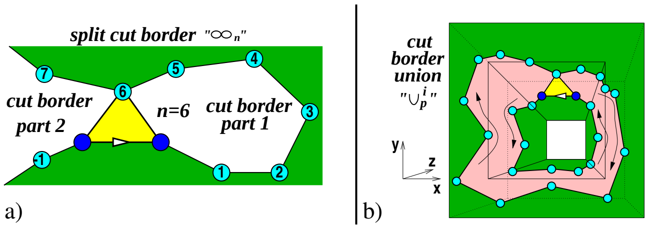

Figure 3: The *"split cut-border"*-/*"cut-border union"*-operation needs one/two indices to specify the third vertex with which the current cut-border edge forms the next triangle. The vertices of the current edge are shaded dark and the newly attached triangle light.

2.1 Traverse Order

Figure 2 illustrates a breadth-first traverse order of a sample triangle mesh. The buffered vertices are connected with bold lines. The collection of these lines is called the *cut-border*. The cut-border divides the triangle mesh into two parts, the so far decompressed part – the *inner part* (shaded) – and the rest – the *outer part*. The edges on the cut-border are enumerated in the order they are processed. Each time a cut-border edge is processed, a new triangle is added to this edge with a basic building operation imposed by the connectivity of the triangle mesh. The new triangle introduces new cut-border edges and the processed edge is removed from the cut-border.

These are the basic rules which define the traverse order. The degrees of freedom lay in the choice of the initial triangle, which constitutes the initial cut-border, and in the way the new edges are enumerated. It will turn out in section 4.2, that the choice of the initial triangle doesn't influence the compression significantly. In the breadth-first strategy the new cut-border edges obtain increasing numbers, such that the cut-border is grown in a cyclic way. A depth-first order is achieved by enumerating the new cut-border edges in a decreasing order, such that the last introduced cut-border edges are processed first. The complete information in the vertex buffer can be used to determine the position of the new cut-border edges. As we concentrate on real time compression, only the strategies which consume no additional computation time are analyzed in section 4.2.

2.2 Building Operations

Let us take a closer look at figure 2 as nearly all basic building operations arise in this small example. The triangle mesh is always built from an initial triangle consisting of the first three vertices. The initial building operation is not encoded but will be denoted with the symbol *"Δ"*. Between figure 2a and 2b to each of the three initial cut-border edges a triangle is attached in the same way as to a triangle strip. Each operation introduces a new vertex and two new edges to the cut-border. Let us call this building operation *"new vertex"* and abbreviate it with the symbol *"$*$"*. The new cut-border edges are enumerated in the order they are added to the cut-border in order to bring about a breadth-first traverse order.

Between figure 2b and 2c the triangle of the outer part, which is adjacent to edge $\mathbf{3}$, is added to the so far (de)compressed triangle mesh. This time no new vertex is inserted, but edge $\mathbf{3}$ forms a triangle with the preceding cut-border edge. This operation will be called *"connect backward"* and is represented by the symbol *"$\leftarrow$"*.

Moving on to figure 2d, two *"new vertex"*-operations arise at the cut-border edges $\mathbf{4}$ and $\mathbf{5}$. At the cut-border edge $\mathbf{6}$ the mirror image of the *"connect backward"*-operation is applied to connect this edge to the subsequent edge. Naturally, this operation is called *"connect forward"* and is abbreviated with *" $\rightarrow$"*. No triangle is added to cut-border edge $\mathbf{9}$ as it is part of the mesh border. This fact has to be encoded, too, and is called *"border"*(*"$\_$ "*)-operation.

A more complex operation appears at cut-border edge $\mathbf{10}$ in figure 2e. The adjacent triangle in the outer part is neither formed with the preceding nor with the subsequent cut-border vertex, but with a vertex further apart. The result is that the cut-border splits into two parts. In figure 2f the first part is formed by the edges $\mathbf{11}$, $\mathbf{12}$ and $\mathbf{16}$ and the second part by $\mathbf{13}$, $\mathbf{14}$ and $\mathbf{15}$. This operation will be called *"split cut-border"* (*"∞_i"*), which takes the index i to specify the third vertex relative to the current cut-border edge. Figure 3a shows another *"split cut-border"*-operation. The relative indices are written into the cut-border vertices. The *"split cut-border"*-operation has two consequences. Firstly, the cut-border cannot be represented anymore by a simple linked list, but by a list of linked lists. And secondly, the choice of the next cut-border part to be processed after a *"split cut-border"*-operation yields a new degree of freedom for the traverse order. To minimize the number of cut-border parts the cut-border part with fewer vertices is chosen.

Another operation arises in figure 2f at cut-border edge $\mathbf{11}$. The adjacent triangle closes the triangle mesh and the current cut-border part is removed. This operation is called *"close cut-border"* and is denoted by *"∇"*. As the size of the current cut-border part is known during compression and decompression, the *"close cut-border"*-operation can also be encoded with *"connect forward"* or *"connect backward"* and the different symbol is only introduced for didactic reasons. On the other hand if there really is a hole in the form of a triangle, three *"border"*-operations are encoded.

Finally, there exists a somewhat inverse operation to the *"split cut-border"*-operation – the *"cut-border union"*-operation. An example is visualized in figure 3b. The figure shows in perspective a cube with a quadratic hole. The so far compressed inner part consists of the two green shaded regions. There are two cut-border parts which are connected by the new yellow triangle, which is attached to the current edge (dark blue vertices). Therefore, this operation is called *"cut-border union"* or for short *"$\bigcup_p^i$"*. Two indices are needed to specify the second cut-border part p and the index i of the vertex within the second cut-border part. The vertices in a cut-border part are enumerated according to the cut-border edges. Therefore, the vertex at the beginning of the cut-border edge with the smallest index in the cut-border part p is numbered zero, the vertex at the second smallest cut-border edge is numbered one and so forth.

op.:	vertex	inner edge	border edge	triangle
Δ	3	0	0	1
$\star$	1	1	0	1
$\rightarrow/\leftarrow$	0	2	0	1
$\_$	0	0	1	0
∞_i	0	1	0	1
∇	0	3	0	1
$\bigcup_p^i$	0	1	0	1

Table 1: The table shows for each basic building operation which mesh elements are finally placed into the inner part.

It can be shown that the number of *"cut-border union"*-operations is exactly the genus of the compressed triangle mesh. Seen from a different angle, the operations "∇", "$\rightarrow/\leftarrow$", "∞_i" and "$\bigcup_p^i$" provide the possibility to connect the current cut-border edge to any possible vertex in the cut-border, whereas the operations "Δ" and "$\star$" utilize new vertices.

2.3 Compressed Representation

The encoding of the sequence of atomic building operations uniquely defines the connectivity of a triangle mesh. The connectivity of the sample mesh in figure 2 can be encoded by the following sequence of operations:

$$\star\star\star\leftarrow\star\star\rightarrow\_\,\infty_2\rightarrow\_\,\_\,\_$$

The symbols for the different operations can be encoded with Huffman Codes to achieve good compression rates. Therefore, the mesh connectivity is sequentially stored in a *bit stream*.

The geometry and material data must be supplied additionally. For each vertex this data can include the vertex position, the surface normal at the vertex and the texture coordinates or some material information. We will refer to all this data with the term *vertex data*. The material of the mesh can also be given for each triangle. Similarly, data can be supplied for the inner edges and the border edges of the mesh. We will collect the different kinds of data in the terms *triangle data*, the *inner edge data* and the *border edge data*. Thus for each type of mesh element, data can be supplied with the connectivity of the mesh. We refer to the collection of all additional data with the term *mesh data*.

Depending on the application there exist two approaches to combine the connectivity and the mesh data of a compressed triangle mesh.

If an application is supplied with enough storage for the complete mesh data, the bit stream for the connectivity can be stored separately. For each type of mesh element the specific data is stored in an array. While the triangle mesh is traversed a vertex, triangle, inner edge and border edge index is incremented after each operation, such that the current mesh elements can be found in the corresponding arrays with the suitable indices. Table 1 shows the increments for each index after the different operations. For example after a *"connect forward"*-operation the inner edge index is incremented by two and the triangle index by one. The advantage of this representation is that the mesh data can be processed without traversing the mesh, for example to apply transformations to the coordinates and normals.

If the compressed triangle mesh is passed to the graphics board or if the mesh data is encoded with variable length values, no random access to the vertex data is possible. Then the mesh data is inserted into the bit stream for the mesh connectivity. After each operation symbol in the stream, the corresponding mesh data is sent to the stream appropriately. For example after a *"split cut border"*-symbol the mesh data for one inner edge and one triangle is transmitted (see table 1). If we only assume vertex and triangle data and denote the vertex data for the i^{th} vertex with v_i and the triangle data for triangle j with t_j, the extended bit stream representation of the mesh in figure 2 would be:

$$v_0v_1v_2t_0\star v_3t_1\star v_4t_2\star v_5t_3\leftarrow t_4\star v_6t_5\star v_7t_6\rightarrow t_7\_\,\infty_2t_8\rightarrow t_9\_\,\_\,\_$$

Remember that the initial triangle is implicitly stored without symbol and introduces the vertices v_0, v_1, v_2 and the triangle t_0.

If the triangle mesh consists of several unconnected components, the compressed bit stream representation consists of the concatenation of the bit streams of the different components.

3 Implementation

All algorithms which process the compressed representation are based on the implementation of the data structure for the cut-border as introduced in section 3.1. This data structure implements the rules which define the traverse order. All other algorithms such as the compression and decompression algorithms presented in the sections 3.2 and 3.3 use this implementation. Further algorithms such as homogeneous transformations of the mesh geometry would also use the cut-border data structure to iterate through the compressed representation

The data structures and algorithms in this section are given in a C++-like syntax. For readability and brevity parentheses were replaced by indentation and additional key-words.

3.1 Cut-Border Data Structure

Data Structure 1 cut border

```
struct Part
    int       rootElement, nrEdges, nrVertices;
struct Element
    int       prev, next;
    Data      data;
    bool      isEdgeBegin;
struct CutBorder
    Part      *parts, *part;
    Element   *elements, *element;
    Element   *emptyElements;

              CutBorder (int maxParts, int maxElems) ;
    bool      atEnd () ;
    void      traverseStep (Data &v0, Data &v1) ;

    void      initial (Data v0, Data v1, Data v2) ;
    void      newVertex (Data v) ;
    Data      connectForward/Backward () ;
    void      border () ;
    Data      splitCutBorder (int i) ;
    Data      cutBorderUnion (int i, int p) ;

    bool      findAndUpdate (Data v, int i, int p) ;
```

The data structure for the cut-border is a list of doubly linked lists storing the vertex data of the buffered vertices. All elements in the doubly linked lists of the different parts are stored within one homogeneous buffer named *elements*. The maximum number of vertices in the buffer during the compression or decompression defines its size. The maximum buffer size is known once the triangle mesh is compressed and can be transmitted in front of the compressed representation. For the first compression of the mesh the maximum number of vertices can be estimated by $10\sqrt{n}$ (see section 4.2), where n is the number of vertices in the triangle mesh. With this approach a simple and efficient memory management as described in [6] is feasible. Only the pointer *emptyElements* is needed, which points to the first of the empty elements in the buffer. Any time a new element is needed, it is taken from the empty elements and the deleted elements are put back to the empty elements. On the one hand the memory management avoids dynamic storage allocation which is not available on graphics boards and on the other hand it speeds up the algorithms by a factor of two if no memory caches influence the performance.

The different parts can be managed with an array *parts* with enough space for the maximum number of parts which are created while the mesh is traversed. Again this number must be estimated for the first compression and can be transmitted in front of the compressed representation. Thus the constructor for the cut-border data structure takes the maximum number of parts and the maximum number of cut-border elements.

part and *element* point to the current part and the current element within the current part respectively. Each part stores the index of its root element, the number of edges and the number of vertices. These numbers may differ as each part is not simply a closed polygon. Any time a *"border"*-operation arises one cut-border edge is eliminated but the adjacent cut-border vertices can only be removed if they are adjacent to two removed edges. Therefore, each cut-border element stores in addition to the indices of the previous and next element and the vertex data, a flag which denotes whether the edge beginning at this cut-border element belongs to the cut-border or not.

The cut-border data structure provides methods to steer the traversal via a bit stream or with the help of a triangle mesh. The methods *atEnd*`()` and *traverseStep*`(&`v_0`, &`v_1`)` are used to form the main loop. The method *traverseStep*`(&`v_0`, &`v_1`)` steps to the next edge in the cut-border data structure and returns the vertex data of the two vertices forming this edge. If no more edges are available, *atEnd*`()` becomes true.

During decompression the operations are read from the bit stream and the cut-border data structure is updated with the corresponding methods *initial*, *newVertex*, *connectForward/Backward*, *border*, *splitCutBorder* and *cutBorderUnion*. For compression additionally the method *findAndUpdate* is needed to localize a vertex within the cut-border data structure. The part index and vertex index are returned and can be used to deduce the current building operation. If the vertex has been found by the *findAndUpdate*-method, it is connected with the current cut-border edge.

3.2 Compression Algorithm

Besides the cut-border we need two further data structures for the compression algorithm — a triangle mesh, with random access to the third vertex of a triangle given a half edge, and a permutation. The random access representation of the triangle mesh provides two methods – the *chooseTriangle*`(`v_0, v_1, v_2`)`-method, which returns the vertex data v_0, v_1, v_2 of the three vertices in an initial triangle, and the method *getVertexData*`(`i_0, i_1`)`, which takes the vertex indices i_0 and i_1 of a half edge v_0v_1 and returns the vertex data of the third vertex of the triangle containing v_0v_1. The permutation is used to build a bijection between the vertex indices in the random access representation and the vertex indices in the compressed representation. It allows to map an index of the first kind to an index of the second kind and to determine whether a certain vertex index in the random access representation has been mapped.

Given a random access triangle mesh, the compression algorithm computes the mentioned permutation and the compressed representation of the mesh, which is sent to a bit stream. The current vertex index of the compressed representation is counted in the index *vertexIdx*. After the initial triangle is processed, the cut-border data structure is used to iterate through the triangle mesh. In each step the vertex data v_0 and v_1 of the current cut-border edge is determined. From the vertex indices the vertex data of the third vertex in the triangle adjacent to the current edge is looked up in the random access triangle mesh. If no triangle is found, a *"border"*-operation is sent to the bit stream. Otherwise it is checked whether the new vertex has already been mapped, i.e. sent to the cut-border. If not, a *"new vertex"*-operation is sent to the bit stream and the vertex index is mapped to the current index in the compressed representation. If the third vertex of the new triangle is contained in the cut-border, the *findAndUpdate*-method is used to determine the part index and the vertex index within that cut-border part. If the part index is greater than zero, a *"cut-border union"*-operation is written. Otherwise a *"connect forward/backward"*-operation or a *"split cut-border"*-operation is written dependent on the vertex index.

Algorithm 1 compression

Input:	*RAM*	...	random access representation
Output:	*bitStream*	...	compressed representation
	perm	...	permutation of the vertices

```
vertexIdx = 3;
RAM.chooseTriangle(v_0, v_1, v_2);
perm.map((v_0.idx, 0), (v_1.idx, 1), (v_2.idx, 2));
bitStream << v_0 << v_1 << v_2;
cutBorder.initial(v_0, v_1, v_2);
while not cutBorder.atEnd() do
    cutBorder.traversalStep(v_0, v_1);
    v_2 = RAM.getVertexData(v_1.idx, v_0.idx);
    if v_2.isUndefined() then
        bitStream << "_";
        cutBorder.border();
    else
        if not perm.isMapped(v_2.idx) then
            bitStream << "*" << v_2;
            cutBorder.newVertex(v_2);
            perm.map((v_2.idx, vertexIdx++));
        else
            cutBorder.findAndUpdate(v_2, i, p);
            if p > 0 then bitStream << "\bigcup_p^i";
            else if i == ±1 then bitStream << "→/←";
            else bitStream << "∞_i";
```

3.3 Decompression Algorithm

The decompression algorithm reads the compressed representation from an input bit stream and enumerates all triangles. The triangles are processed with the subroutine *handle*`(`v_0,v_1,v_2`)`, which for example renders the triangles. As in the compression algorithm, firstly, the initial triangle is processed and then the mesh is re-built with the help of the cut-border methods *atEnd* and *traversalStep*. In each step the next operation is read from the bit stream and the corresponding method of the cut-border data structure is called such that the third vertex of the new triangle is determined in order to send it to the subroutine *handle*.

4 Measurements and Optimizations

In this section we analyze our software implementation of the compression and decompression algorithm. Firstly, we introduce the

Algorithm 2 decompression

Input: *bitStream* ... compressed representation
Output: *handle* ... processes triangles

bitStream $>> v_0 >> v_1 >> v_2$;
handle (v_0, v_1, v_2);
cutBorder.initial (v_0, v_1, v_2);
while not *cutBorder.atEnd*() do
  cutBorder.traversalStep (v_0, v_1);
  bitStream >> *operation*;
  switch (*operation*)
    case "$\rightarrow$/$\leftarrow$":
      handle $(v_1, v_0,$
        cutBorder.connectForward/Backward()$)$;
    case "∞_i":
      handle $(v_1, v_0,$ *cutBorder.splitCutBorder*(i)$)$;
    case "$\bigcup_p^i$":
      handle $(v_1, v_0,$ *cutBorder.cutBorderUnion*(i,p)$)$;
    case "*":
      bitStream $>> v_2$;
      cutBorder.newVertex (v_2);
      handle (v_1, v_0, v_2);
    case "_":
      cutBorder.border();

triangle mesh				compr	decom	storage
name	**t**	**n**	$\lvert$**bd**$\rvert$	$k\Delta/s$	$k\Delta/s$	$bits/t$
genus5	144	64	0	386	782	4.23±5.7%
vase	180	97	12	511	796	2.15±6.0%
club	515	262	6	541	857	2.09±3.5%
surface	2449	1340	213	490	790	1.87±0.8%
spock	3525	1779	30	496	820	1.97±0.7%
face	24118	12530	940	430	791	1.81±0.3%
jaw	77692	38918	148	332	809	1.62±0.5%
head	391098	196386	1865	321	796	1.71±0.1%

Table 2: The basic characteristics of the models, the compression and decompression speed and the storage needs per triangle.

test set of models in section 4.1. Then we examine the influence of the traverse order on the compression ratio and the size of the cut-border (section 4.2). And finally we gather the important results on the performance of the presented algorithms in section 4.3.

4.1 The Models

The measurements were performed on the models shown in figure 4. All models are simple connected 2-manifolds and differ in their size. From top left: genus5, vase, club, surface, spock, jaw, face, head. The detail of the head model is hidden in the small interior structures. Therefore, we present in figure 4 a view into the inside of the head.

The basic characteristics of the models are shown in the left half of table 2. Here the number of triangles $\mathbf{t}$, the number of vertices $\mathbf{n}$ and the number of border edges $|\mathbf{bd}|$ are tabulated for each model.

4.2 Traverse Order and Cut-Border Size

In section 2.1 we defined the traverse order up to the choice of the initial triangle and the enumeration of newly introduced cut-border edges. To study the influence of the initial triangle we measured the storage needs for the compressed connectivity of each model

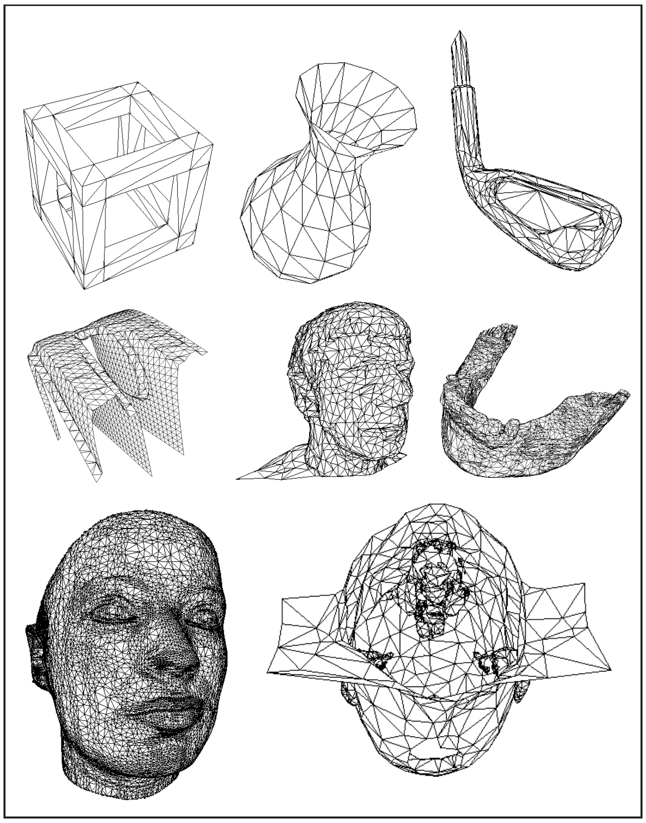

Figure 4: The models used to analyze the compression and decompression algorithms.

several times with randomly chosen initial triangles. Then we computed for each model the average value and the standard deviation as tabulated on the very right of table 2. The influence of the initial triangle vanishes with increasing size of the model and is still less than ten percent for the smallest models. With the same measurements the fluctuation of the cut-border size was determined as shown in table 3. Here the fluctuation is higher and reaches up to twenty percent for the jaw and the club models.

There are a large number of enumeration strategies for the newly introduced cut-border edges. For performance reasons and the simplicity of the implementation, we favored the enumeration strategies which can be implicitly handled with the cut-border data structure introduced in section 3.1. Therefore a newly introduced cut-border edge may either be delayed until all present edges are processed or the new edge is processed next. These two strategies apply to the *"connect forward/backward"*-operations and correspond to attaching the next highest and the next smallest edge index respectively to the new edge. In the case of a *"new vertex"*-operation two new edges are introduced to the cut-border. In this case three possible strategies are feasible. Either the first/second new edge is processed next or both edges are delayed. The *"split cut-border"*-and the *"cut-border union"*-operations arise much more rarely and therefore were excluded from the analysis of the traversal strategy. Thus we were left with twelve strategies, three choices for the *"new vertex"*-operation and for each *"connect"*-operation two choices. Luckily, it turned out that the strategy, where the new edge is processed next after both *"connect"*-operations and where the second edge is processed next after a *"new vertex"*-operation, is superior over all others. This strategy achieved best compression and kept the cut-border smallest for all models.

Table 3 shows for each model the maximum number of parts and the maximum number of buffered vertices needed for mesh traver-

name	$part_{\max}$	$vert_{\max}$	$prop$
genus5	3.21±12.7%	32.75±15.4%	5.35
vase	2.30±24.2%	22.56±10.2%	2.99
club	3.11±11.9%	44.24±21.0%	4.45
surface	3.10± 9.7%	83.16±12.1%	3.10
spock	3.24±13.2%	120.10± 4.5%	3.23
face	3.40±15.6%	329.08±14.5%	4.22
jaw	4.76±10.7%	564.42±19.7%	4.55
head	9.00±11.1%	1255.20± 8.6%	3.56

Table 3: The maximum number of parts and the maximum number of buffered vertices needed for mesh traversal. The last column gives the quantity $prop = (vert_{\max} + 6 \cdot s_{vert}) / \sqrt{n}$.

sal. The values are averaged over several random choices of the initial triangle. As the values fluctuate significantly we add three standard deviations to the values such that 99.73% of the values are smaller than our estimation if we suppose a normal distribution. The maximum number of cut-border parts is comparably small and can safely be estimated by 100 for the first compression of a triangle mesh. To show that the maximum number of buffered vertices increases with $\sqrt{n}$ we divide the measured values plus three standard deviations by $\sqrt{n}$ and get values between three and six independent of the size of the model. Thus a save estimation for the size of the vertex buffer in a first compression of a triangle mesh is $10\sqrt{n}$.

4.3 Performance

The last column of table 2 shows that our approach allows compression of the connectivity of a triangle mesh to two bits per triangle[3] and less. The theoretical lower limit is 1.5 bits per triangle which is achieved with uniform triangle meshes. To understand this fact let us neglect the *"split cut-border"*- and *"cut-border union"*-operations. Each *"new vertex"*-operation introduces one vertex and one triangle, whereas each *"connect"*-operation only introduces a triangle to the mesh. To arrive at a mesh with twice as many triangles as vertices, equally many *"new vertex"*- and *"connect"*-operations must appear. The Huffman code for the *"new vertex"*-operation consumes one bit and the *"connect"*-operations are encoded with two and three bits as still other operations must be encoded. If both *"connect"*-operations are equally likely, we get a compression to 1.75 bits per triangle. If on the other hand one *"connect"*-operation is completely negligible a compression to 1.5 bits is feasible. The optimal traversal strategy found in the previous section avoids *"connect backward"*-operation and therefore allows for better Huffman-encoding than the other strategies.

Table 2 also shows the compression and decompression speed in thousands of triangles per second measured on a 175MHz SGI/O2 R10000. The decompression algorithm clearly performs in linear time in the number of triangles with about 800,000 triangles per second. But the performance of the compression algorithm seems to decrease with increasing n. Actually, this impression is caused by the 1 MB data cache of the O2 which cannot keep the complete random access representation of the larger models, whereas the small cut-border data structure nicely fits into the cache during decompression. On machines without data cache the performance of the compression algorithm is also independent of n. The compression algorithm is approximately half as fast as the decompression algorithm. About 40% of the compression time is consumed by the random access representation of the triangle mesh in order to find the third vertex of the current vertex. The other ten percent are used to determine the part and vertex index within the cut-border.

[3]The genus5 model consumes more storage as its genus forces five *"cut-border union"*-operations and the model is relatively small.

If our compression scheme is used to increase the bandwidth of transmission, storage or rendering, we can easily compute the break-even point of the bandwidth. The total time consumed by our compression scheme is the sum of the times spent for compression, transmission and decompression. The total time must be compared to the transmission time of the uncompressed mesh. Let us assume for the uncompressed representation an index size of 2 bytes, such that each triangle is encoded in 6 bytes. If we further use the estimation that the triangle mesh contains twice as many triangles as vertices, the break-even point computes[4] to a bandwidth of 12MBit/sec. Thus the compression scheme can be used to improve transmission of triangle meshes over standard 10MBit Ethernet lines. As our approach allows us to compress and decompress the triangle mesh incrementally, the triangle mesh can also be compressed and decompressed in parallel to the transmission process. Then even the transmission over a 100MBit Ethernet line could be improved.

5 Extensions

In this section we describe how to extend our method on non orientable triangle meshes. Additionally, we show how to encode attributes which are attached to vertex-triangle pairs.

5.1 Non Orientable Triangle Meshes

As we restricted ourselves to 2-manifold triangle meshes, the neighborhood of each vertex must still be orientable even for non orientable meshes. From this follows that each cut-border part must be orientable at any time: a cut-border part is a closed loop of adjacent edges. The orientation of one edge is passed on to an adjacent edge through the consistent orientation of the neighborhood of their common vertex. Therefore, only the *"split cut-border"*- and *"cut-border union"*-operations need to be extended as they introduce or eliminate cut-border parts. Both operations connect the current cut-border edge to a third vertex in the cut-border, which is either in the same or in another cut-border part. The only thing which can be different in a non orientable triangle mesh is that the orientation of the cut-border around this third vertex is in the opposite direction as in the orientable case. Therefore, only one additional bit is needed for each "∞_i"- and "$\bigcup_p^i$"-operation, which encodes whether the orientation around the third vertex is different. During compression the value of the additional bit can be checked from the neighborhood of the third vertex. Previously a *"split cut-border"*-operation produced a new cut-border part. In the new case with different orientations around the third vertex, the orientation of one of the new parts must be reversed and the parts are concatenated again as illustrated in figure 5. In a *"cut-border union"*-operation the cut-border part containing the third vertex is concatenated to the current cut-border part and in the new case the orientation of the cut-border part with the third vertex is reversed before concatenation.

5.2 Attributes For Vertex-Triangle Pairs

A lot of triangle meshes contain discontinuities that force attachment of certain vertex attributes to vertex-triangle pairs. See for example the genus5 model in figure 4, which contains a lot of creases. For each vertex on a crease exist two or more different normals which must be attached to the same vertex which is contained in different triangles. Thus for models with creases it must be possible to store several different vertex normals for different vertex-triangle pairs. Similarly, discontinuities in the color attribute force storage

[4]with a compression rate of 400,000 triangles per second and a decompression rate of 800,000 triangles per second

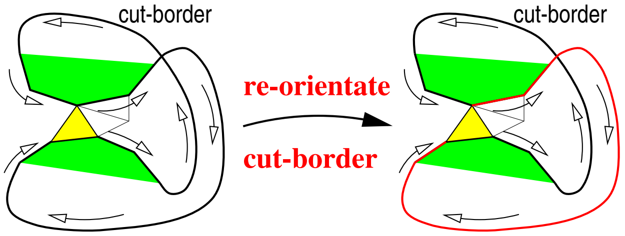

Figure 5: After some *"split cut-border"*-operations of a non orientable manifold half of the cut-border (drawn in red) must be reoriented and no new part is generated.

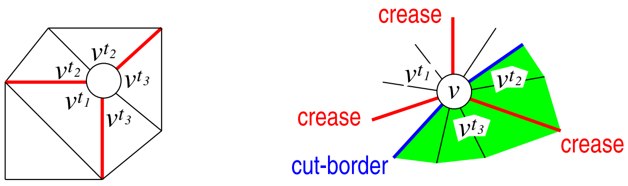

Figure 6: Creases divide the neighborhood of a vertex into regions. Each region contains the triangles with one vertex-triangle attribute.

of several RGBA values within the vertex-triangle pairs. A simple solution to support vertex-triangle attributes is to encode these attributes with every appearance of a vertex-triangle pair. This implies that the same vertex-triangle attributes for one vertex may be replicated several times. On the other hand if we duplicated these vertices, which lay on creases, the vertex coordinates would be replicated.

With a small amount of overhead we can do better and encode each vertex location and each vertex-triangle attribute exactly once. Let us denote the collection of all vertex specific data as for example its coordinates with v and the different collections of the vertex-triangle data with $v^{t_1}, v^{t_2}, \ldots$. As an example let us describe the encoding of v^t in the case of creases as illustrated in figure 6. The neighborhood of each vertex is split by the creases into several regions. Within each region there is exactly one vertex-triangle attribute valid for the vertex and all triangles in this region. On the right side of figure 6 a cut-border vertex is shown during compression or decompression. We see that at any time it is sufficient to store besides the vertex data v two vertex-triangle attributes $v^{t_{\text{left}}}$ and $v^{t_{\text{right}}}$ for each vertex within the cut-border. When a new triangle is added to the cut-border, the vertex-triangle attributes of a vertex can only change, if the vertex is part of the current cut-border edge and if this edge is a crease. If one of the vertex-triangle attributes for example $v^{t_{\text{left}}}$ changes after an operation which adds a triangle, there are two possible cases. Either a new vertex-triangle attribute is transmitted over the bit stream or the new attribute is copied from $v^{t_{\text{right}}}$.

To encode when a new vertex-triangle attribute has to be transmitted we transmit one or two control bits after each operation, which adds a triangle to the current cut-border edge. Two control bits are needed only for the *"connect"*-operations as the new triangle contains two cut-border edges. The control bits encode whether the affected cut-border edges are creases. Afterwards, for each cut-border vertex on a cut-border edge, which is a crease, we transmit one further bit which encodes whether a new vertex-triangle attribute is transmitted or the attribute should be copied from the other vertex-triangle attribute stored in the cut-border. If we denote the total number of inner edges with e and the total number of crease edges with e_c this approach results in an overhead of less than $e + 2e_c$ bits.

6 Conclusion

The presented compression technique provides not only a fast decompression algorithm but also a compression algorithm which performs in only double the amount of time of the decompression algorithm. The slow down of the compression algorithm is primarily caused by the uncompressed random access mesh representation. Faster mesh data structures must be investigated as well as the usage of hash tables to speed up the search for vertices within the cut-border.

The simplicity of the algorithms allow for hardware implementation and suitable hardware will be designed in future work. But also software implementations perform extremely well as shown in the previous section. Beside the good performance and the simplicity of the approach the connectivity of a triangle mesh is compressed similarly well as by the best known compression methods. Therefore even globally optimizing compression algorithms can be replaced by the faster and simpler approach.

Acknowledgments

Many thanks to Reinhard Klein and Andreas Schilling for inspiring discussions and to Michael Doggett for reviewing the paper.

References

[1] E. M. Arkin, M. Held, J. S. B. Mitchell, and S. S. Skiena. Hamiltonian triangulations for fast rendering. *Lecture Notes in Computer Science*, 855:36–57, 1994.

[2] Rueven Bar-Yehuda and Craig Gotsman. Time/space tradeoffs for polygon mesh rendering. *ACM Transactions on Graphics*, 15(2):141–152, April 1996.

[3] M. Deering. Geometry compression. In *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 13–20, 1995.

[4] Francine Evans, Steven S. Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In *IEEE Visualization '96*. IEEE, October 1996. ISBN 0-89791-864-9.

[5] Silicon Graphics Inc. GL programming guide. 1991.

[6] Scott Meyers. *Effective C++: 50 specific ways to improve your programs and designs. – 2. ed.* Addison-Wesley, Reading, MA, USA, 1997.

[7] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide — The Official Guide to Learning OpenGL, Version 1.1.* Addison-Wesley, Reading, MA, USA, 1997.

[8] Gabriel Taubin and Jarek Rossignac. Geometric compression through topological surgery. Technical report, Yorktown Heights, NY 10598, January 1996. IBM Research Report RC 20340.