

# Tree-based Data Structures for Triangle Mesh Connectivity Encoding

Ioannis Ivrissimtzis, Christian Rössl, and Hans-Peter Seidel

Max-Planck-Institut für Informatik,  
Stuhlsatzenhausweg 85, Saarbrücken, 66123, GERMANY  
{*ivrissim, roessler, hpseidel*}@mpi-sb.mpg.de

## 1 Introduction

Triangle meshes have recently emerged as the de facto standard in many Computer Graphics applications, generating a research interest in finding data structures able to represent them efficiently. This is not a trivial task given that the size of a typical meshes can vary from few hundreds triangles, up to hundreds of millions of triangles for some very detailed models.

A typical data set describing a 3D triangle mesh model consists of connectivity, geometry and some external attributes. The connectivity describes the way the vertices of the mesh are connected with edges and faces. It captures intrinsic topological properties of the mesh, containing all the information related to the genus and the existence and size of boundaries. The geometry of the mesh describes the actual positions of the vertices in the 3-dimensional Euclidean space  $\mathbf{R}^3$ . Finally, there are the other attributes of the mesh, like color or normals. In this paper we deal with the encoding of connectivity.

The choice of the most suitable data structure for connectivity encoding usually depends on the characteristics of the application. If the mesh is going to be manipulated heavily, then the focus is on the efficient traversal and access of the vertices, edges and triangles. Therefore, the most suitable data structures, like for example the winged-edge data structure, have a lot of redundant information. For rendering purposes more compact representations are preferred. The usual choice is a simple shared vertex representation encoding triangles as triplets of indices into a vertex table. The *ply*-format, supported also by the Stanford 3D Scanning Repository, is a well known example of such a representation. For storage and efficient transmission of the mesh over low-bandwidth networks as the Internet the focus is on the further compression of the data.

In this paper we propose a novel data structure for encoding triangle mesh connectivity, consisting of a binary tree with positive integers attached on its nodes. It is simple enough so that it can be implemented easily, and compact enough so that it can be used for compression purposes. It arises naturally from a Divide and Conquer Algorithm which we will describe in detail. Notice that in the compression literature the focus is almost solely on the algorithms



themselves, and some of the most widely used methods do not use any data structure other than a stream of symbols.

The main advantage of introducing and studying this data structure is that we can find a simple criterion for deciding whether a member of the data structures represents a valid mesh or not. This criterion can be used directly to check the validity of a code without going into the reconstruction of the mesh, or indirectly for enhancing the compression ratio. Criteria of this kind are rather rare in the literature, because if the encoding is very compact usually the corresponding algorithm is very involved, while otherwise it is very difficult to deal with the redundant information.

### 1.1 Overview

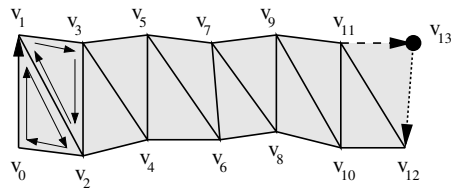
In Section 2 we give some basic standard terminology. In Section 3 we give a brief overview of the existing encoding techniques. In Section 4 we describe in detail the Divide and Conquer Algorithm, mainly following [12]. Finally, in Section 5 we study the data structure acquired by the Divide and Conquer Algorithm, showing that we can achieve very good compression ratios and also obtain an interesting new insight into the complex mathematics of triangle meshes.

## 2 Terminology

Some basic terminology we use throughout the paper.

The *Rooted Triangle Mesh* is a triangle mesh, with one directed edge on the boundary marked. The marked edge is called the *gate*. If the mesh has no boundary we create one by removing one triangle. Usually we will refer to the rooted triangle meshes simply as meshes, and we will make the distinction between rooted and unrooted meshes only when it is necessary. Also, notice that the same distinction can be carried over to the trees. A *tree* is defined as an acyclic graph and the *rooted tree* as a tree with one node, the *root*, marked. Again, it is customary to refer to the rooted trees simply as trees if there is no room for confusion.

A *zig-zag strip* of length  $n$  is a mesh with  $n + 2$  vertices  $(v_0, v_1, \dots, v_{n+1})$  and  $n$  triangles  $v_i v_{i+1} v_{i+2}$  with  $0 \leq i \leq n - 1$  (cf. Fig. 1). In the rooted zig-



**Fig. 1.** A zig-zag strip of length 12. The gate is marked with a solid thick arrow, the left leading directed edge is dashed, the right is dotted, and the leading vertex is marked by a black dot.



zag strips we use here the gate will always be the directed edge  $(v_0, v_1)$ . The vertex  $v_{n+1}$  will be called the *leading vertex* and the triangle  $v_{n-1}v_nv_{n+1}$  the leading triangle. Assuming that all the triangles have the same orientation, as indicated by the gate, the oriented leading triangle is  $(v_{n-1}, v_n, v_{n+1})$  for odd  $n$  and  $(v_n, v_{n-1}, v_{n+1})$  for even  $n$ . For odd  $n$ ,  $(v_n, v_{n+1})$  and  $(v_{n+1}, v_{n-1})$  are the left and the right directed leading edges, respectively. For even  $n$  the left and right leading directed edges are  $(v_{n-1}, v_{n+1})$  and  $(v_{n+1}, v_n)$ , respectively. In both cases the left leading directed edge is the one pointing at the leading vertex.

### 3 Overview of Mesh Encoding Algorithms

In the last years the increasing popularity of triangle meshes has rapidly accelerated the pace of research in the area of mesh encoding. As a result, many diverse techniques have emerged for the encoding of triangle mesh connectivity, each one with some advantages over all the others when a particular class of meshes is considered. Some of the earlier techniques include the encoding of the connectivity as a permutation of the vertices [5], the topological surgery method [21], where a mesh is encoded as a vertex tree together with the dual face tree, and the pioneering Cut Border Machine [10], [8], which was the first recursive method based on a traversal of the triangles of the mesh.

More recently two major branches of encoding techniques have started to emerge. The valence driven techniques, where the algorithm traverses the vertices of the mesh, and the EdgeBreaker like methods, where the algorithm traverses the faces of the mesh. In the valence driven methods, each vertex transmits its valence together with some special symbols. The method was initiated by Touma-Gotsman in [22], with a non-adaptive traversal of the mesh vertices, while Alliez-Desbrun used an adaptive traversal of the vertices [1], reporting the best compression ratios in practice. Alliez and Desbrun's algorithm has been expanded to more complex problems such as the progressive transmission of a 3D model including the geometry [2], or the encoding of polygonal meshes [14], giving again the best reported results when compared with any other similar method.

The other major branch of recently developed techniques is based on the EdgeBreaker algorithm. The original EdgeBreaker was proposed in [19]. It traverses a tree of the faces of a mesh and for each face returns one symbol from an alphabet of five, determining the adjacencies of that face with the not yet conquered part of the mesh. Its main advantage over the valence driven methods is the existence of a sharp guaranteed worst-case bound, which originally was reported at 4 bits per vertex, and later was improved to 3.67 bits per vertex [15], and to 3.55 bits per vertex [9]. Numerous other improvements in the efficiency of the technique followed. For example, [18] make the encoding and decoding process linear in time, [11] simplify further



the decoding algorithm especially for meshes of arbitrary topology, while [20] is an adaptation of the EdgeBreaker for highly regular meshes. The Divide and Conquer algorithm we study here traverses the faces of the mesh and can be classified into the family of the EdgeBreaker like schemes.

At a more theoretical level, we notice that some of the mathematical foundations of the triangle mesh connectivity encoding were laid much earlier, in 1962, with the results of Tutte [23] on the enumeration of the planar rooted triangulations. Tutte found generating functions for the number of distinct rooted triangulations over a plane, and studied their asymptotic behavior. His results established a theoretical upper bound of 3.24 bits per vertex for the encoding of sufficiently large triangle meshes. In [4], Tutte's initial condition that two boundary vertices cannot be connected with a non-boundary edge is removed and it is proven that the generating functions have the same asymptotic behavior.

Finally, in [6], a method employing an approach very similar to ours was reported for graph compression. There, a graph is decimated with the use of graph separators, that is, subgraphs whose removal separates the graph into components of roughly similar size, and in a recursive process they encode these components in a tree data structure. The different setting, especially the nature of the graph as a combinatorial rather than a geometric object, makes much more difficult conclusive answers, something that, as we see in this paper, is not the case with triangle meshes.

## 4 The Divide and Conquer Algorithm

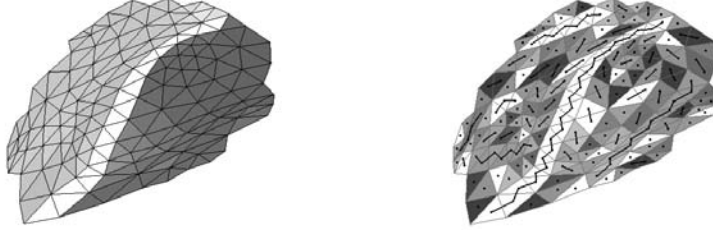
In this section we describe in detail the Divide and Conquer algorithm. The algorithm was introduced in [12] where some additional details can be found. We first describe the algorithm for planar meshes, then we study some basic characteristics of its behavior, and show the necessary modifications for the handling of arbitrary topology. We conclude the section with a brief discussion of the internal similarities between our algorithm and the EdgeBreaker.

### 4.1 The Divide and Conquer Algorithm for Planar Meshes

In the beginning of the encoding process we mark the boundary vertices of the mesh as conquered. We randomly choose a directed edge on the boundary of the mesh as the initial gate and we build a zig-zag strip conquering its vertices. We stop when we arrive at an already conquered vertex. The latter happens when the leading vertex of the zig-zag strip reaches either the boundary of the mesh or another vertex of the strip. In both cases the original rooted mesh splits into two rooted submeshes: The left submesh with the left leading directed edge of the strip as gate, and the right submesh with the right leading edge of the strip as gate. Any of these submeshes or both can be empty. We continue recursively, encoding separately the two submeshes and the encoding



process terminates when all the submeshes are empty. A typical situation is shown in Fig. 2. We organize the data acquired in this process in the form



**Fig. 2.** A triangle strip divides each mesh into two submeshes (left). The left and the right submeshes are processed the same way recursively defining a binary tree. The result is shown on the right side, the black lines denote the strip connectivity. We can encode a planar triangle mesh as the resulting binary tree with only the strip lengths stored in its nodes.

of a binary tree with the strip lengths stored in its nodes. The length of the initial zig-zag strip is stored in the root of the tree, the encoding of the left submesh is the left branch of the tree, and the encoding of the right submesh is the right branch of the tree.

Encoding a triangle mesh can be written in pseudo code as:

```
tree = encode(edge) {
    node.length=0;
    while (!conquered(oppositeVertex(edge)) { // GROW
        conquer(oppositeVertex(edge), triangle(edge));
        edge=nextStripEdge(edge);
        ++node.length;
    }
    if (node.length==0)
        return NULL; // STOP
    node.left =encode(leftLeading(edge)); // RECURSION
    node.right=encode(rightLeading(edge));
    return node;
}
```

The encoding starts with the enter gate as argument to `encode()`, e.g.  $(v_0, v_1)$  in Fig. 1. `oppositeVertex()` returns the “leading” vertex opposite of the `edge` in the same triangle, e.g.  $v_2 \in \triangle(v_0, v_1, v_2)$ . `nextStripEdge()` finds the next edge in the next triangle of the strip, e.g.  $(v_2, v_1) \in \triangle(v_2, v_1, v_3)$ , and `leftLeading()/rightLeading()` return the adjacent edges in the same triangle, e.g.  $(v_1, v_2), (v_2, v_0)$ . Navigation in the mesh can be reduced to several calls to `next(edge, orientation)` and `neighbor(edge)` operations returning the next edge and the neighboring directed edge in the adjacent triangle.

Conversely, in one recursive step of the decoding process we have a zig-zag strip of specified length and two rooted meshes, and we glue them together. We identify the gate of the left rooted mesh with the left leading edge of the strip, and we glue the left boundary of the strip with the boundary of the



mesh stopping at the gate of the strip. Then we repeat the same for the right rooted mesh and the right boundary of the strip. The gate of the new mesh is the gate of the strip.

Assume that the tree encoding the mesh is traversed in preorder. Then the recursive decoding can be sketched as follows:

```

decode() {
  length=getNextNode(); // READ, preorder traversal
  if (length>0) { // else: leaf, STOP
    [ enter,leading[2] ]=createStrip(length);
    enterSub[0]=decode(); // RECURSION
    enterSub[1]=decode();
    for (i=0;i<2;++i) // GLUING
      if (enterSub[i]!=EMPTY)
        glue(leading[i],enterSub[i]);
    return enter;
  }
  return EMPTY;
}

```

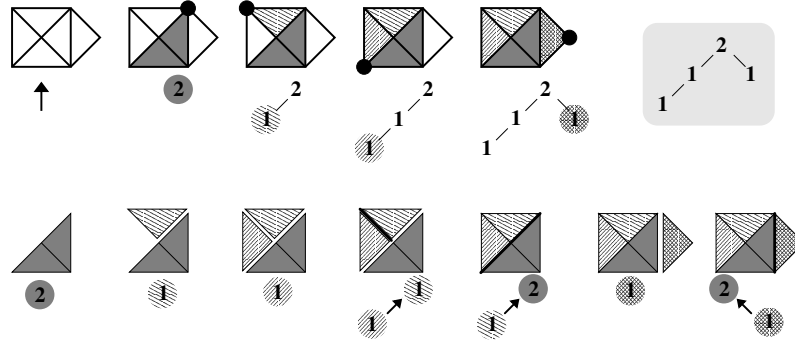
Here, `getNextNode()` returns the values of the tree nodes, `createStrip()` creates a triangle strip and returns its `enter` gate/edge, e.g.  $(v_0, v_1)$ , in Fig. 1, and the two `leading[]` edges, e.g.  $(v_{11}, v_{13}), (v_{13}, v_{12})$ . The recursive call to `decode()` creates the two submeshes and returns their enter gates. Finally, the strip boundaries are glued to the submeshes. The number of boundary edges for `glue()` can easily be calculated from the strip's length and parity, e.g. 6 and 7 for the strip of length 12. Note that some modification is needed to handle the self intersection, valence 3 and non-planar topology cases.

E.g. in Fig. 2 (left) we glue starting from the leading edges (top right in the picture) first the blue and then the green submesh to the dividing strip (red). In both cases gluing stops at the gate of the strip (bottom left). Fig. 3 shows an example run of the algorithm for a simple mesh.

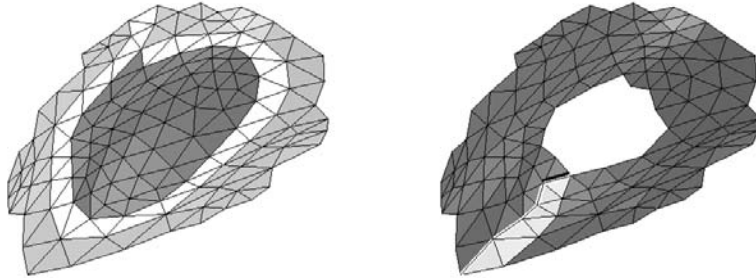
Note that a strip may intersect not only with the boundary of a submesh but also with itself resulting in a loop (cf. Fig. 4, left). This is the same situation as before but with an inner and an outer submesh. The leading vertex now induces some kind of singularity for gluing: When we start gluing from the inner leading edge (red-blue border) we cannot decide which direction to take once we arrive back at the leading vertex as the outer boundary has not been glued yet. This problem can be resolved by gluing both leading edges first (`glueSingleEdges(leading[1],enterSub[1])` removes the singularity) before gluing along the whole strip (`glue(leading[0],enterSub[0])`).

Another fine point of the algorithm is the occurrence of empty submeshes. A strip of length 1, that is a single triangle, can have one or both submeshes empty if one or both leading edges are boundary edges of the mesh. A strip of length 2 can only have the left submesh empty because the non-gate vertex of the first triangle of the strip is not boundary. Otherwise the process of growing the strip would stop there. In a strip of length greater than 2 both the leading directed edges are not boundary and an empty submesh occurs only when the strip intersects itself and the internal submesh is empty. This happens precisely when the strip passes through a vertex of valence 3. The



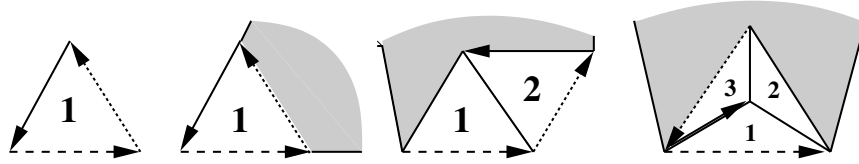


**Fig. 3.** Example run of the algorithm. *Top row:* The mesh (top left) is recursively encoded as a binary tree (top right) starting from the gate shown by the arrow. The four resulting strips are filled with different patterns. The first strip of length 2 partitions the mesh into two submeshes, all other strips have one or two empty submeshes. The dot denotes the leading vertex of the current strip, i.e. the point where the strip touches a conquered/boundary vertex. For every new strip the corresponding tree state is shown with the new node highlighted. *Bottom row:* For decoding, the tree is traversed in preorder. The strips are created during the top-down traversal of the tree nodes, and submeshes are glued (thick line) when an edge of the tree is followed in bottom-up direction. In the figure the corresponding tree nodes and edges are shown below the state of the mesh.



**Fig. 4.** Left: A strip may intersect itself resulting in a loop. Right: Handles or holes prevent the strip from splitting the mesh into two parts. A special *split* code is output instead of the empty left submesh. It references the corresponding edge in the right submesh drawn black. With this information the remaining boundary (white) can be glued.





**Fig. 5.** From left to right. All strips are entered through the dashed gate (bottom line). The first strip of length 1 has both submeshes empty, while the next strip of length 1 has only the right submesh empty (solid gate). The strip of length 2 has the left submesh empty (dotted gate). The strip of length 3 has the right submesh empty (solid gate). Notice that in the case of a strip with length greater than 2 the empty submesh is in the interior of the strip and not on the boundary (valence 3 case).

parity of the length of the strip determines if the right or the left mesh is empty (cf. Fig. 5).

This case has to be checked separately, as the empty submesh does not correspond to boundary. If the strip `length > 2` and the submesh `k` is empty (`enterSub[k] == EMPTY`) then an extra glue operation is needed: `glue(stripEdge, enterSub[k])`, where `stripEdge` is the corresponding edge of the strip (cf. triangle 1 in Fig. 5, right) which can be referenced easily by a fixed navigation path back through the strip.

## 4.2 Analysis of the Divide and Conquer Algorithm

Next, we give two basic propositions concerning the behavior of the Divide and Conquer Algorithm. The first proposition is about the size of the obtained binary tree, and the second describes the relationship between the structure of the tree and the strip length information stored in its nodes.

The number of vertices of the mesh will be denoted by  $v$ , the number of triangles of the mesh, which is equal to the sum of the strip lengths, will be denoted by  $t$ , and the number of the strips, which is equal to the number of the nodes of the tree, will be denoted by  $n$ . We have

**Proposition 5.** *Let  $n$  be the number of the strips obtained from the divide and conquer algorithm. We have  $n = v - 2$ , where  $v$  is the number of the vertices of the mesh.*

**Proof:** We notice that a strip of length  $l$  conquers  $l - 1$  vertices, and thus  $n$  strips of total length  $t$  conquer  $t - n$  vertices. The number of vertices to be conquered is  $v_I$ , the number of interior vertices of the mesh, hence, we have  $t - n = v_I$ , giving  $n = t - v_I$ . The latter can also be written  $n = v - 2$  as we immediately see by applying Euler's formula.  $\square$



The binary tree and the strip length information can be combined in a single data structure, namely a binary tree with positive integer weights assigned to its nodes. The next question is when such a data structure is valid, i.e. when it represents a planar triangle mesh. We have

**Proposition 6.** *Let  $T$  be a tree with  $n$  nodes and let the sum of the strip lengths be  $t$ . Then,  $T$  represents a planar mesh if and only if*

- (i)  $t + 1 \leq 2n$  holds for  $T$  and all its subtrees.
- (ii) *The strip lengths of the nodes with only left child are odd, and the strip lengths of the nodes with only right child are either 1 or even.*

**Proof:** Let  $b$  be the number of the boundary vertices of a planar mesh. Applying Euler's formula gives

$$b = 2v - t - 2 \quad (1)$$

which from Proposition 5 becomes

$$b = 2n - t + 2 \quad (2)$$

Then, the inequality  $3 \leq b$ , holding for the boundary of the mesh, gives  $t + 1 \leq 2n$ . Notice that the inequality  $3 \leq b$  is a standard assumption in the literature of meshes as  $b = 2$  would give two boundary vertices connected between them with two different edges, while  $b = 1$  would give a vertex connected with itself.

Condition (ii) describes the exceptional cases where one of the submeshes is empty and two edges of the dividing strip are glued together.

Conversely, if the condition (i) of the proposition holds, there is enough free boundary to perform all the gluing operations and we get a valid triangle mesh, while condition (ii) guarantees that we can perform the gluing in the exceptional cases as well.  $\square$

### 4.3 Arbitrary Topology

In the case of arbitrary topology the main difference is that a dividing strip can have the same non-empty submesh on the left and the right. The simplest example is a planar mesh with a hole in it, which is topologically equivalent to a cylinder (cf. Fig. 4, right). In this case we need to encode only one branch of the tree and give some additional information on how the boundary of the corresponding submesh is glued to the other side of the strip.

If the genus of the mesh is  $g$  and there are  $h$  holes in it, it is a simple topological fact that the number of the strips which do not separate the mesh is at most  $2g + h$ . For each such strip we need  $O(2 \log v)$  bits to identify the directed edge for gluing, and in the worst case  $O(\frac{1}{2} \log v)$  bits for the extra symbol.



#### 4.4 The Connection with the EdgeBreaker

Before the study of the obtained data structure, it is worth having a look at the algorithm in a more general setting, clarifying some aspects which might be obscured in a very concrete exposition. A first observation is that the algorithm, like the EdgeBreaker, implicitly induces a traversal of the triangles of the mesh. This traversal of the triangles can be seen at two levels, the first level is the traversal of the triangles of a zig-zag strip, from the root of the strip to the leading edges, and the second level is the traversal of the tree that stores these zig-zag strips.

Also, we notice that the algorithm works not only with zig-zag strips but with general strips as well. In fact, any bitstream the encoder and the decoder would agree on, defines a way of building general strips, and thus a variation of the method. Here we use the zig-zag strips as the most natural choice for making a strip.

The fan strip is another important class of strips. In this case, assuming a preorder traversal of the tree, we get the same traversal of the triangles as the EdgeBreaker, and our approach differs only in the interpretation of the obtained data. In fact, we can translate the encoding of the binary tree and the strip lengths into the familiar  $C, L, E, R, S$  string of the EdgeBreaker and vice-versa. Each strip length  $n$  can be written as a string of  $n - 1$   $C$ 's, and for each node of the tree we use one of the  $L, R, S, E$  symbols, depending on whether it has only left child, only right child, two children or no child. Notice that because our data structure is non-linear, namely, a binary tree rather than a symbol stream, it is not necessary to assume any particular traversal of the tree to interpret the data. Although this has advantages in the theoretical analysis of the algorithm, nevertheless, in the implementation we usually assume a traversal to make things simpler.

### 5 Tree-based Data Structures

With the Divide and Conquer Algorithm, the encoding of triangle mesh connectivity encoding can be split into two separate but closely related subtasks. The encoding of the binary tree, and the encoding of the strip lengths stored in its nodes. We study these two encodings, first separately and then in their interrelation. For simplicity we deal with planar meshes only. But, taking into account the special output symbols and the changes of the Euler's characteristic they introduce, the arbitrary topology can be treated in a similar manner.

#### 5.1 Binary Tree Encodings

The binary tree is one of the most popular structures for storage and maintenance of data, and thus, many basic related problems have been widely



studied. For a brief comparative study of different binary tree encodings see [13] and [16].

In our context, the relevance of the tree encoding methods becomes apparent with the observation that any binary tree can correspond to a planar mesh. Indeed, if all the strip lengths are equal to 1, then by Proposition 6 every binary tree gives a valid planar mesh. The number of all the binary trees with  $n$  nodes is given by the Catalan number  $C_n$

$$C_n = \frac{(2n)!}{n!(n+1)!} \sim \frac{4^n}{\sqrt{\pi}n^{3/2}} \quad (3)$$

see [7] Exercise 9.8. Thus, there is an asymptotic bound of 2 bits per node.

Some of the existing methods for binary tree encoding enumerate all the trees with  $n$  nodes so that each tree can be represented by an integer number. Obviously, such methods achieve optimal compression ratios, but the encoding and decoding cost is very high, and especially for the large trees we use here this cost is prohibitive. Other encoding methods traverse the tree, transmitting one letter for each node. These methods are separated into two categories. The methods which use a fixed alphabet and those using an alphabet depending on the number of the nodes  $n$ .

The most common fixed alphabet encodement uses 4 letters, let say  $\{L, R, S, E\}$  to make the analogy with the EdgeBreaker clearer. Each letter determines if the corresponding node has only left, only right, both or none children. The cost is  $2n$  bits, which, asymptotically achieves the bound of 2 bits per node given by (3). By Proposition 5 this is also equal to 2 bits/v.

Another well-known fixed alphabet encoding is the Zaks' sequences which use the two letters alphabet  $\{0, 1\}$ . The encoding process first transforms the binary tree into a complete binary tree by appending new leaves wherever possible, that is, two new leaves at any old leaf and one new leaf at any single child node. Then we traverse the nodes of the complete binary tree in preorder, transmitting a 1 if the node is internal and a 0 if the node is a leaf. There are standard algorithms deciding when a given sequence of 0's and 1's is the Zaks' sequence of a tree.

The size of the file encoding the tree can be further reduced with Arithmetic Coding [17]. In this case the compression ratio also depends on the traversal of the tree. Notice that there are traversals, like the inorder, which do not work with the above four or two letter encodings, because we can not reconstruct the tree without some additional information.

The variable alphabet encodements now, most often use the letters of the set  $\{0, 1, \dots, n\}$ . When it comes to compression issues they have the problem that it is more difficult to find a sharp guaranteed upper bound which can be trivially found for a fixed alphabet. Nevertheless, many times they are more flexible, and there are standard algorithms determining the validity of a code, making it easy to eliminate the transmission of redundant information.

An example of variable length encoding is the weight sequence, see [16]. There, the letter corresponding to a node is the number of the nodes of its



left subtree. The inductive argument showing that the method works is that if we know the number of the nodes of the tree, and the number of the nodes of the left subtree we can find the number of nodes of the right subtree by subtraction, and we can continue recursively this process until reaching the leaves. The weight sequence encoding of a tree has a special interest because by Proposition 5 the number of the nodes  $n$  is related to  $v$  the number of the vertices of the mesh. For this reason, the weight sequence is a useful intermediate representation of a binary tree corresponding to a mesh.

## 5.2 Strip Lengths Encodings

The encoding of the strip lengths is equivalent to the encoding of  $v - 2$  numbers summing up to  $t$ . The range of the numbers is from 1 up to the length of the largest strip we create. Although the largest strip has always length less than  $\lfloor \frac{v}{2} \rfloor$ , still this is not a sharp bound with any practical use.

The simplest way to encode such a sequence of numbers is to represent the number  $k$  as a word of  $k$  bits consisting of  $k - 1$  1's followed by a 0. The total cost in this case is  $t$ , and the total cost of encoding the mesh, including the tree, is bounded by  $t + 2v - 4 < 4v$ . It worth noticing here that the guaranteed performance of 4 bits/ $v$  of this very coarse encoding method, can not be improved without going deeper into the study of the relationship between the encoding tree and the corresponding strip-lengths.

Another, very popular method to encode a sequence of numbers is the Huffman coding. Each number is assigned a unique code and the number of bits we spend on each code depends on the probability of each number to appear in the sequence. After the Huffman coding we can use Arithmetic Coding to further exploit any existing entropy. Notice that the compression achieved with the Arithmetic Coding depends on the order the strip lengths are transmitted, that is, on the particular traversal of the tree. Also, notice that because now the tree is given, and unlike the situation in Subsection 5.1, any traversal works. For a survey of different tree traversals, see [3].

If the mesh has a relatively small boundary the number of vertices, and thus, the number of tree nodes is about half the number of triangles. Therefore, the average strip length is near 2, and the entropy of strip lengths largely depends on the number of strips with length 2. We have noticed by experiment that the more regular a mesh the more strips of length 2 occur in its encoding. For an intuitive explanation of the last, see Figure 2: a large strip passing through a regular area of the mesh creates a regular boundary with vertices of valence 4, and this creates a lot of strips with length 2. This observation partly justifies the choice of the leading directed edges of the dividing strip as the gates of the two submeshes. Another deterministic choice of the gates, for example near the middle of the dividing strip, would increase the length of the strips near the root of the tree but the result would be worse compression ratios, because the total entropy would decrease.



The Table 1 shows the experimental results for preorder and postorder encoding of the tree with a 4 and a 2 letter alphabet, and preorder, inorder, postorder and level traversal in the encoding of the strip lengths, for a variety of meshes. In [12] we used preorder traversals for both the tree and the strip lengths and we transmitted them in an interwoven fashion, that is, at each node the code of the strip length followed immediately after the tree code. Comparing the results there, even with the most favorable combination of separate transmission of tree and strip lengths file, we see that they are better. That means that there is a lot of entropy in the blend of tree codes and strip lengths, and this entropy is exploited by the Arithmetic Coder.

Mesh	#V	#F	$t_{pre,1}$	$t_{pre,2}$	$t_{post,2}$	$s_{pre,H}$	$s_{pre}$	$s_{post}$	$s_{in}$	$s_{level}$	IRS02
david1	315	586	2.46	2.46	2.46	1.40	2.08	2.08	2.03	2.08	3.94
david2	1512	2924	2.11	2.10	2.11	1.43	1.66	1.67	1.66	1.67	3.56
david3	6035	11820	2.03	2.03	2.03	1.46	1.58	1.60	1.53	1.62	3.39
david4	24085	47753	2.05	2.06	2.06	1.46	1.51	1.53	1.39	1.55	3.18
dinosaure	14070	28136	2.04	2.06	2.05	1.45	1.52	1.54	1.37	1.55	3.13
fandisk	6475	12946	1.52	1.50	1.50	1.51	1.23	1.24	0.75	1.57	1.95
mannequin1	428	839	2.34	2.34	2.34	1.36	1.89	1.91	1.87	1.91	3.79
mannequin2	11703	23402	0.92	0.91	0.90	1.30	0.84	0.83	0.34	1.21	1.06
venus	8268	16532	2.04	2.05	2.05	1.49	1.60	1.61	1.55	1.63	3.46
max-planck	100086	199996	1.20	1.16	1.16	1.52	1.08	1.06	0.67	1.39	1.42

**Table 1.** Compression ratios in bits/vertex for 10 models from [1]. The trees ( $t$ ) are encoded separately from the strip lengths ( $s$ ). The suffixes indicate pre-, post-, in-, and level traversal, the 1-alphabet (Zaks) or 2-alphabet. For  $s_{pre,H}$  only Huffman coding is applied, Arithmetic Coding and a fixed alphabet is used in all other columns. The last column shows the results in [12].

Next we assume that the tree code and the strip length code of a node are sent one after the other. That means that we assume the same traversal for both the tree code and the strip lengths code. We study the relationship between the two encodings, and we see how we can save information from the tree code using information from the strip-lengths and vice versa.

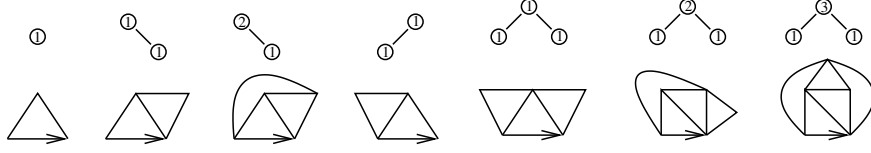
### 5.3 Tree First Transmission

Here we assume that the code of the tree is sent first, using the  $\{L, R, S, E\}$  alphabet, and the code of the strip length follows. Some simple observations can reduce the amount of information we have to send for the strip lengths.

A first observation is that because of Proposition 6, condition (ii), the weight of an  $R$  node is either 1 or an even number, while the weight of an  $L$  node is always odd. Another observation, from the same Proposition, condition (i), is that all the leaves have strip length 1, and do not need encoding. Going one step further, we notice that if an  $R$  is just above a leaf then, by condition (i), the corresponding strip length is either 1 or 2 and can be encoded with a single bit. Similarly, an  $L$  node just above a leaf can only



have a strip length 1, and we do not need to transmit any information. Fig. 6 shows the encoding near the leaves and the corresponding meshes.



**Fig. 6.** The encoding trees and the corresponding meshes near the leaves. Notice that some of the rooted meshes are isomorphic as unrooted meshes.

The above observations are the simplest instances of a more general feature of our approach, namely that every node of the tree represents a gluing operation and Proposition 6 gives a simple criterion to determine when such a gluing operation is legal. Therefore, we can treat these instances in a unified way by determining at each node all the legal gluing operations, and sending only the necessary information for the decoder to distinguish between them. This assumes a postorder reconstruction of the mesh.

Checking the criterion of the special gluings is straightforward. By Proposition 6 condition (ii), if the node is an  $L$  then the set of possible strip lengths is restricted to the odd numbers, while if the node is an  $R$  the set of possible strip lengths consists of the even numbers and the 1. For a criterion checking the regular gluings, i.e. Proposition 6 condition (i), for each node we need the number of nodes of its subtree, and the sum of the strip lengths of that subtree. This information can be held in an auxiliary data structure, where each node  $\mathbf{n}_j$  of the tree is assigned two integers  $(n_j, w_j)$ , the number of nodes and the sum of strip lengths of its subtree. Notice that the integers  $w_j$  are essentially the weight sequence of the tree, see Subsection 5.1. When we process a new node  $\mathbf{n}_j$ , we first find its children, let say  $\mathbf{n}_{j_l}$ ,  $\mathbf{n}_{j_r}$ , and if the integers assigned to them are  $(n_{j_l}, w_{j_l})$  and  $(n_{j_r}, w_{j_r})$ , then

$$n_j = n_{j_l} + n_{j_r} + 1 \quad \text{and} \quad w_j = w_{j_l} + w_{j_r} + s \quad (4)$$

where  $s$  is the strip length corresponding to  $\mathbf{n}_j$ . Then, Proposition 6, condition (i), gives

$$w_{j_l} + w_{j_r} + s + 1 \leq 2(n_{j_l} + n_{j_r} + 1) \quad (5)$$

giving

$$s \leq 2n_{j_l} + 2n_{j_r} - w_{j_l} - w_{j_r} + 1 \quad (6)$$

This way we find explicitly the set of all the strip lengths giving a legal gluing operation at a node, and instead of transmitting the actual strip length we send an offset determining its position in the set of all legal values. If the



set contains only one element, that is, when the set is the  $\{1\}$  we send no information. If the set has two elements, that is, if it is either the  $\{1, 2\}$  or the  $\{1, 3\}$  we send only one bit. In all the other cases it is better to resort to Huffman encoding of the offsets rather than using an ad hoc code for every particular set, because of the significantly higher frequencies of the short strips.

The algorithm we just described can also be used in conjunction with the original Divide and Conquer algorithm, where we transmit the actual strip lengths, as a test checking the validity of a code. In this case, proceeding as above, we find the set of all legal strip lengths corresponding to a node and we check if the actual value of  $s$  lies into that set. If this happens for every node then all the gluing operations are legal and the code describes a valid mesh. If there is a node with strip length outside the set, then the decoding process will break at that point. Of course the testing algorithm must be coupled with an algorithm checking the legality of the corresponding tree code, which can be found in the literature.

#### 5.4 Strip-lengths First Transmission

Suppose now that we first transmit the strip-length of a node and then the tree code corresponding to it. From Proposition 6, condition (ii), even strip lengths correspond to an  $S$  or an  $R$  node, while odd strip lengths greater than 1 correspond to an  $S$  or an  $L$  node. Therefore, we need a single bit for the tree code of the nodes with strip length greater than 1.

Table 2 shows some more results obtained with the encoding techniques described in Subsections 5.3, 5.4. An additional column compares to the Edgebreaker-like traversal when fan-strips are used instead of zig-zag strips. We also show the results obtained by the Alliez-Desbrun method [1], which currently gives the best compression ratios.

#### 5.5 The Valence 3 Vertices

From Proposition 6, and the above discussion of its implications, it is apparent that the valence 3 vertices are very characteristic. Their peculiarity arises from the fact that a zig-zag strip collapses to itself only when passing through a valence 3 vertex. Equivalently, it is the only case when a strip of length greater than 2 can correspond to an  $R$  or an  $L$  node. Therefore, in many cases it may pay off to have an initial preprocessing step clearing the mesh from its valence 3 vertices. Such a strategy to improve the efficiency of an algorithm was also proposed in [2].

After the clearance step we work as in Subsections 5.3, 5.4 separating the case of tree code transmission first from the case of strip length transmission first. Sending the tree code first, we know that an  $L$  node can only store a strip length equal to 1, because any greater strip length would create a valence 3



Mesh	pre	post,H	post	fan	A&D
david1	3.94	3.30	3.94	3.86	2.96
david2	3.53	3.35	3.53	3.57	2.88
david3	3.36	3.40	3.36	3.37	2.70
david4	3.13	3.39	3.14	3.19	2.52
dinosaure	3.09	3.38	3.10	3.13	2.25
fandisk	1.90	3.42	1.94	1.94	1.02
mannequin1	3.79	3.29	3.78	3.79	2.51
mannequin2	1.02	3.27	1.07	1.05	0.37
venus	3.42	3.40	3.42	3.45	2.37
max-planck	1.38	3.43	1.42	1.42	n/a



**Table 2.** Compression ratios in bits/vertex. The table shows results for transmitting the tree (2-alphabet) and indices of valid strip lengths in an interwoven fashion in pre- and post- order with Huffman coding only resp. Arithmetic Coding. The *fan* column gives results from using not zig-zag strips but fan-strips and a preorder traversal, fixed alphabeth which corresponds to the bitstream obtained from the Edgebreaker (with Arithmetic Coding applied). The compression ratios are similar. The right most column shows the results of Alliez-Desbrun [1] for comparison. In this paper renderings of the corresponding meshes can be found. The Max-Planck mesh with 100086 vertices has also been used for testing.

vertex, and therefore, we do not need to send any strip length information. Similarly, a strip length corresponding to an  $R$  is either 1 or 2, and is encoded in a single bit. On the other hand, if we first transmit the strip length of a node, then any length greater than 2 corresponds to an  $S$  node and we do not need any extra tree code. A strip length equal to 2, corresponds to either an  $S$  or an  $R$  node and we need a single bit for the tree code.

## 6 Conclusion

We described a Divide and Conquer algorithm for the encoding of triangle mesh connectivity. The naturally arising data structure for the storage of the obtained information is a binary tree with positive integer numbers assigned to its nodes. We studied this data structure, showing that there is a deep correlation between the structure of the tree and the assigned integers, which can benefit the performance of the algorithm.

## References

1. P. Alliez and M. Desbrun. Valence-Driven connectivity encoding for 3D meshes. In *EUROGRAPHICS 01 Conference Proceedings*, pages 480–489, 2001.
2. Pierre Alliez and Mathieu Desbrun. Progressive compression for lossless transmission of triangle meshes. In *SIGGRAPH 01, Conference Proceedings*, pages 195–202, 2001.



3. Alfs Bertiss. A taxonomy of binary tree traversals. *BIT*, 26:266–276, 1986.
4. W.G. Brown. Enumeration of triangulations of the disk. *Proc. Lond. Math. Soc., III. Ser.*, 14:746–768, 1964.
5. M. Denny and C. Sohler. Encoding a triangulation as a permutation of its point set. In *Proceedings of the 9th Canadian Conference on Computational Geometry*, pages 39–43, May 15–17 1997.
6. N. Deo and B. Litow. A structural approach to graph compression. In *MFCSS Workshop on Communications*, pages 91–101, 1998.
7. Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics: a foundation for computer science. 2nd ed.* Addison-Wesley, 1994.
8. Stefan Gumhold. Improved cut-border machine for triangle mesh compression. In *Erlangen Workshop '99 on Vision, Modeling and Visualization, IEEE Signal Processing Society*, 1999.
9. Stefan Gumhold. New bounds on the encoding of planar triangulations. Technical Report WSI-2000-1, Wilhelm-Schikard-Institut für Informatik, Tübingen, March 2000.
10. Stefan Gumhold and Wolfgang Straßer. Real time compression of triangle mesh connectivity. In *SIGGRAPH 98 Conference Proceedings*, pages 133–140, July 1998.
11. Isenburg and Snoeyink. Spirale reversi: Reverse decoding of the edgebreaker encoding. *CGTA: Computational Geometry: Theory and Applications*, 20, 2001.
12. I. Ivriissimtzis, C. Rössl, and H-P. Seidel. A divide and conquer algorithm for triangle mesh connectivity encoding. In *Pacific Graphics 02, Conference Proceedings*, 2002.
13. Jyrki Katajainen and Erkki Mäkinen. Tree compression and optimization with applications. *Int. J. Found. Comput. Sci.*, 1(4):425–447, 1990.
14. Andrei Khodakovsky, Pierre Alliez, Mathieu Desbrun, and Peter Schröder. Near-optimal connectivity encoding of 2-manifold polygon meshes. *Graphical Models, special issue on compression*, 2002.
15. Davis King and Jarek Rossignac. Guaranteed 3.67V bit encoding of planar triangle graphs. In *Proceedings of the 11th Canadian Conference on Computational Geometry*, pages 146–149, 1999.
16. E. Mäkinen. A survey in binary tree codings. *Comput. J.*, 34(5):438–443, 1991.
17. Moffat, Neal, and Witten. Arithmetic coding revisited. *ACMTOIS: ACM Transactions on (Office) Information Systems*, 16, 1998.
18. Rossignac and Szymczak. Wrap&zip decompression of the connectivity of triangle meshes compressed with edgebreaker. *CGTA: Computational Geometry: Theory and Applications*, 14, 1999.
19. Jarek Rossignac. Edgebreaker: Connectivity compression for triangle meshes. In *IEEE Transactions on Visualization and Computer Graphics*, volume 5 (1), pages 47–61. 1999.
20. Szymczak, King, and Rossignac. An edgebreaker-based efficient compression scheme for regular meshes. *CGTA: Computational Geometry: Theory and Applications*, 20, 2001.
21. Gabriel Taubin and Jarek Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, April 1998.
22. Costa Touma and Craig Gotsman. Triangle mesh compression. In *Proceedings of the 24th Conference on Graphics Interface (GI-98)*, pages 26–34, June 18–20 1998.
23. W.T. Tutte. A census of planar triangulations. *Can. J. Math.*, 14:21–38, 1962.



<http://www.springer.com/978-3-540-40116-2>

Geometric Modeling for Scientific Visualization

Brunnett, G.; Hamann, B.; Müller, H.; Linsen, L. (Eds.)

2004, IX, 488 p., Hardcover

ISBN: 978-3-540-40116-2