

# Fast and Effective Stripification of Polygonal Surface Models

Xinyu Xiang\*    Martin Held†    Joseph S. B. Mitchell‡

Department of Applied Mathematics and Statistics  
State University of New York, Stony Brook, NY 11794-3600

## Abstract

A fundamental algorithmic problem in computer graphics is that of computing a succinct encoding of a triangulation of a polygonal surface model in order to be able to transmit and render it efficiently. The goal is to take a given polygonal surface model, whose facets are given by (possibly multiply-connected) polygons, triangulate its facets, and then decompose the triangulation into a small number of “tristrips,” each of which has its connectivity stored implicitly in the ordering of the data points. We develop methods that are effective in solving the stripification problem, both in theory (provably good encodings) and in practice. Our methods are based on carefully constructed search trees in the dual graph, followed by algorithms to decompose dual trees into tristrips. One decomposition algorithm is provably optimal (based on dynamic programming), allowing us a sound basis of comparison among our other (heuristic) algorithms. We demonstrate the speed and effectiveness of our algorithms through a battery of experiments. In comparison with the recently released STRIPE system for stripification, we find that our stripifier, FTSG, produces comparable or better quality encodings, while requiring significantly less computing time on a large variety of datasets. Further, FTSG is carefully engineered and implemented to be robust, even in the face of highly degenerate and corrupted real-world data.

**CR Categories:** I.3.3 [Computer Graphics]: Picture/Image Generation—displaying algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithm, languages, and systems.

## 1 Introduction

We consider a problem of importance in graphics and visualization: computing a succinct encoding of a triangulation of a polyhedral model in order to be able to transmit and render it efficiently.

CAD/CAM and virtual environments applications often require that very complex datasets be visualized at real-time rates. Current 3D graphics rendering hardware often faces a memory bus band-

width bottleneck in the processor-to-graphics pipeline. One naturally wants to avoid rendering unnecessary triangles (e.g., through visibility culling). Also, it is common to simplify and approximate complex models. But it is also important to minimize the time needed to transmit  $n$  triangles that are to be rendered, e.g., by compressing the geometric and topological information in a model, transmitting the compressed data, and decompressing at the rendering stage, hopefully using a very small on-chip cache.

A common encoding method is based on “tristrips,” which permit a 2-vertex cache: when vertex  $v_{i+2}$  is transmitted, it determines a triangle together with the cache vertices  $v_i, v_{i+1}$ . A *tristrip* is an ordered sequence (with repetitions) of vertices,  $(v_1, \dots, v_m)$ , which *encodes* a set of  $m - 2$  triangles. A *sequential* tristrip encodes the set of triangles  $\{(v_i, v_{i+1}, v_{i+2})\}$ ,  $1 \leq i \leq m - 2$ ; a *fan* tristrip encodes the set  $\{(v_1, v_{i+1}, v_{i+2})\}$ , all of which contain  $v_1$ . Fig. 1 gives a sequential tristrip encoding of 12 triangles.

In an ideal encoding, by a single tristrip, only  $n + 2$  (rather than  $3n$ ) vertices would have to be transmitted for a triangulation with  $n$  triangles. In general, if we are able to decompose a surface triangulation into  $k$  tristrips, we will need only  $n + 2k$  vertices.

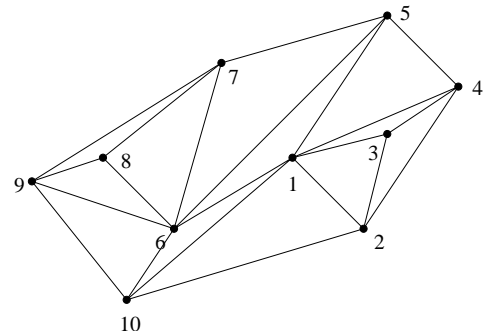


Figure 1: The triangulation is encoded using one tristrip: (1,2,3,4,1,5,6,7,8,9,6,10,1,2).

## Our Contributions

- (1) An efficient (provably linear-time) *robust* algorithm for decomposing a polygonal model into a small number of tristrips. The stripification algorithm is a three-phase method, based on (i) computing a spanning tree of the dual graph of a triangulation, using variants of breadth-first and depth-first searching, (ii) partitioning a tree into tristrips, and (iii) performing a concatenation phase, to join small strips into larger strips.
- (2) A linear-time algorithm, based on dynamic-programming, for *optimally* performing phase (ii), extracting the minimum number of sequential tristrips from a given spanning tree.
- (3) A theoretical analysis of the number of vertices required, in the worst case, to encode a triangulation using tristrips.

\*xxiang@ams.sunysb.edu

†held@cosy.sbg.ac.at. Also with Institut für Computerwissenschaften, Universität Salzburg, Salzburg, Austria.

‡jsbm@ams.sunysb.edu

- (4) Experimental analysis of our algorithms on a wide variety of sample datasets, showing that our methods compare quite favorably with a leading publicly available stripification system (STRIFE) [8]. We obtain orders of magnitude of speed-up over the published version of STRIFE, and our algorithms remain significantly faster than the newly released (unpublished) STRIFE 2.0. In terms of compression, our methods also improve upon STRIFE, decreasing, on average, the number of vertices required to encode a triangulation. We also compare favorably with the “tomesht” utility provided by SGI. Furthermore, our algorithm is *robust* in its ability to handle a wide variety of real-world datasets.
- (5) Our results are the first to study systematically the effect of allowing both sequential and fan tristrips.

## Related Work

There have been several works on constructing tristrips. Arkin et al. [2] study sequential triangulations of point sets and polygons. They show that testing whether a given triangulation of a point set or polygon is sequential can be done in linear time. They also prove that certain non-degenerate point sets in the plane do not admit a sequential triangulation, and that the related problem of computing a Hamiltonian triangulation is  $\mathcal{NP}$ -hard for polygons with holes.

A program that SGI develops in [1] produces generalized tristrips. Its heuristic tends to create strips that begin and end on faces with fewest number of neighbors in the triangulation, so as to reduce the number of isolated triangles. Starting from one of the triangles with fewest neighbors, a greedy algorithm chooses, from its neighbors, the triangle with the fewest number of neighbors as the next triangle. If there is a tie, the program looks ahead one step to check neighbor’s neighbors for fewest number of neighbors. If a tie occurs again, the strip continues in an arbitrary valid way.

Another recently implemented stripifier is “STRIFE” [7]. It adopts SGI’s heuristic described above as its local algorithm. However, for data that contains quadrangles, STRIFE uses a global approach (“patchification”) to construct long strips from large patches of quadrangles. Their implementation claims to handle models containing *convex* polygonal faces.

Our method is perhaps most closely related to that of Speckmann and Snoeyink [11], who implemented an algorithm specially designed for TIN (Triangulated Irregular Network) models. They employ a unique traversal algorithm to find the spanning trees of the dual graphs of TIN models. The spanning trees need never be explicitly determined, but are traversed in the greedy process of finding *sequential* strips. They also have a “cleanup” phase in which leftover single triangles are combined with longer tristrips.

Deering [5] considers *generalized triangle meshes*, having a cache for  $> 2$  vertices, to facilitate geometry compression; particularly efficient algorithms for obtaining such compressions are given by Chow [4]. Bar-Yehuda and Gotsman [3] use planar separator theorems to show that a vertex cache of size  $\Theta(\sqrt{n})$  is sufficient and sometimes necessary to attain optimal transmission (one vertex per triangle). Gumhold and Straßer [9] also develop a connectivity compression/decompression method, using a cache that permits one vertex per triangle transmitted.

There has been recent work leading to impressive results on the compression, of both connectivity and geometry, of manifold triangulation meshes; see, e.g., Taubin et al. [12, 13]. Their compression is not based on stripification, so does not produce the tristrips that we desire; however, their “Topological Surgery” technique, which employs a spanning tree of the vertices, leads to connectivity compressions approaching less than 2 bits per triangle. Touma and Gotsman [14] developed another method that achieves a substantial improvement, using less than 1.5 bits per vertex, on average, for connectivity compression. We note that these non-tristrip methods

of compression do not have the feature that, on average, each vertex is transmitted twice, as we have with tristrips: even if a triangulation can be encoded with a single tristrip (one vertex per triangle), it will require roughly  $2n$  vertices in the encoding, assuming that every triangle has 3 neighboring triangles, since there are  $2n - 4$  triangles in a triangulated planar graph on  $n$  vertices.

SGI’s *tomesht* was initially designed to construct generalized tristrips. It does not hesitate to use “swaps” to make longer strips. The cost is one bit per “swap” in Irix GL, but is one vertex per “swap” in OpenGL. As a result, a generalized tristrip containing  $n$  triangles may be encoded with far more than  $n + 2$  vertices. STRIFE inherits the same problem and the implementation runs relatively slow in practice. Speckmann and Snoeyink’s implementation is fast and memory-efficient. However, it is designed for TIN models, and it usually transmits 3–5% more vertices than STRIFE does, since it produces a larger number of strips [11].

Denny and Sohler [6] consider a related problem of encoding a triangulation by a permutation of its vertex set, by means of constructing a hierarchical family of decimated triangulations, with the order of the vertices determining the reconstruction of the triangulation; their goal, however, was not to obtain tristrips.

## 2 Preliminaries

We begin with some basic definitions. A *polygonal surface model*,  $S$ , is a set of *polygonal faces* that represent the boundary of a polyhedral object in 3-space. Each *polygonal face* is represented by a circular list of *vertices* that describe the outer boundary of the face, followed by a possibly empty list of *holes*. A polygonal face with no holes is said to be *simply connected*, while a face with one or more holes is *multiply connected*. Each polygonal face is usually expected to lie in a plane and each has an associated outward surface normal. Note, however, that real-world data will often have non-coplanar vertices.

The input to our algorithms is a general polygonal surface model  $S$ . We use an enhancement of FIST [10] to triangulate (robustly) each face of  $S$ . FIST is carefully designed to be able to handle arbitrarily nonplanar or corrupted polygonal data in a reasonable manner. Thus, for most of our discussion, we will assume that  $S$  is a *triangulated model* (or a *triangulation*), meaning that every one of its faces is simply a triangle.

In general,  $S$  may consist of several connected components; our analysis will apply to each component separately, so we can assume that  $S$  is connected from now on.

A *manifold edge* is an edge contained in exactly two incident triangles, which are called *adjacent*, while a *non-manifold edge* is contained in more than two incident triangles. A *boundary edge* only belongs to one triangle. In our implementation, a *non-manifold edge* will be split and its incident triangles are paired and distributed such that each instance of the edge becomes either a *manifold edge* or a *boundary edge*. We define the *adjacency graph*,  $\mathcal{G}$ , of a triangulated model  $S$  that contains no non-manifold edges, to be the graph whose nodes are the faces (triangles) of  $S$  and whose edges link pairs of adjacent faces. We see that the degree of each node of  $\mathcal{G}$  is at most three. If  $S$  is a *2-manifold*, meaning that every point of the surface has a neighborhood homeomorphic to a 2-disk, then  $\mathcal{G}$  is a *3-regular graph* (every node has degree exactly three).

We adopt some of the definitions from Arkin et al. [2]. A triangulation  $S$  is *Hamiltonian* if the graph  $\mathcal{G}$  has a Hamiltonian path, meaning that there exists a path in  $\mathcal{G}$  that visits each node (triangle of  $S$ ) exactly once. A triangulation  $S$  is *sequential* if the graph  $\mathcal{G}$  has a Hamiltonian path such that no three consecutive edges crossed by the path are incident on a common vertex. Equivalently,  $S$  is sequential if there exists a vertex sequence,  $\sigma = (v_1, v_2, \dots, v_{n+2})$ , possibly with repetitions, such that the  $n$  triangles of  $S$  are exactly given by the triples of consecutive vertices in the listing:

$(v_1, v_2, v_3), (v_2, v_3, v_4), \dots, (v_n, v_{n+1}, v_{n+2})$ . A triangulation  $S$  is *fan* if the graph  $\mathcal{G}$  has a Hamiltonian path such that all edges crossed by the path are incident on a common vertex.

In some cases (e.g., Iris GL), it is useful to define a more general notion of “sequential”, in which we permit *swaps*, special marks (bits) within a vertex sequence to indicate an exchange of the two cache vertices. Alternatively (e.g., OpenGL), a swap can be effected by sending a zero-area triangle (transmitting an extra vertex instead of a “swap” bit). For example,  $(1, 2, 3, \text{swap}, 4, 5)$  yields triangles  $((1, 2, 3), (3, 2, 4), (2, 4, 5))$ , which also result from the sequence  $(1, 2, 3, 2, 4, 5)$  (having zero-area triangle  $(2, 3, 2)$ ). Another way to achieve this result is to allow *duplicate* triangles to be encoded in the strip:  $(1, 2, 3, 4, 2, 5)$ , which repeats the triangle  $(2, 3, 4)$  (as  $(3, 4, 2)$ ). A sequential triangulation is *pure* if the sequence contains no zero-area or duplicated triangles.

A *tristrip*  $\sigma$  is an ordered sequence (with repetitions) of vertices,  $(v_1, \dots, v_m)$ , which *encodes* a set of  $m - 2$  triangles. A *sequential* tristrip encodes the set of triangles  $\{(v_i, v_{i+1}, v_{i+2})\}$ ,  $1 \leq i \leq m - 2$ ; a *fan* tristrip encodes the set  $\{(v_1, v_{i+1}, v_{i+2})\}$ , all of which contain  $v_1$ . With a slight abuse of notation, we refer both to the vertex sequence and to the encoded set of triangles as the “tristrip”. Thus, a triangulation  $S$  is *sequential* (resp., *fan*) if its triangles are encoded by a single sequential (resp., fan) tristrip.

We are interested in partitioning  $S$  into a small number,  $k$ , of tristrips,  $\{\sigma_1, \dots, \sigma_k\}$ , which *encode*  $S$ . Associated with such a partitioning of  $S$  into tristrips there is an *encoding cost*. Typically, we take the encoding cost to be the number of vertices in the encoding (i.e., the sum of the lengths of the tristrips).

### 3 Theoretical Analysis

Our goal in this section is to give guaranteed bounds on the function  $\mathcal{C}_s(n)$ , which gives the worst-case encoding cost for pure sequential strips as a function of the number  $n$  of triangles in a triangulation  $S$ . We define the cost  $\mathcal{C}'_s(n)$  similarly for sequential strips that allow zero-area triangles, and  $\mathcal{C}_{s,f}(n)$  for encodings using both sequential and fan strips. Using our standard definition of encoding cost in terms of the total number of vertices, we see that we need  $n + 2k^*$  vertices for a given triangulation  $S$ , where  $k^*$  denotes the minimum number of tristrips in any partitioning of  $S$  into (pure) sequential tristrips.

**Lemma 1** *Any (connected) triangulation  $S$  consisting of exactly three triangles is sequential; thus,  $\mathcal{C}_s(3) = 5$ .*

*Proof.* Clearly, if  $S$  is connected and has exactly three triangles, it is Hamiltonian and is topologically equivalent to the configuration shown in Fig. 2(a), which can be encoded with the single sequential strip  $(1, 2, 3, 4, 5)$ .  $\square$

**Lemma 2** *Let  $S$  be a Hamiltonian triangulation consisting of four triangles. Then  $S$  is either sequential or fan; thus,  $\mathcal{C}_{s,f}(4) = 6$ . Further,  $\mathcal{C}_s(4) = 8$  and  $\mathcal{C}'_s(4) = 7$ .*

*Proof.*  $S$  must have the structure of either (b) or (c) in Fig. 2. One (case (b)) can be encoded sequentially, and the other (case (c)) can be encoded as a *fan* strip. Thus,  $\mathcal{C}_{s,f}(4) = 6$ . If we must use purely sequential strips, then in case (c) we are forced to have two strips, resulting in an encoding cost of 8 (instead of 6, for case (a)). If we allow zero-area triangles, then case (c) results in an encoding using only 7 vertices:  $(2, 3, 1, 4, 1, 5, 6)$ . (Alternatively, one can encode it with 6 vertices and a swap:  $(2, 3, 1, 4, \text{swap}, 5, 6)$ .)  $\square$

Consider now a triangulation  $S$ , having  $n$  triangles, that is Hamiltonian, with Hamiltonian path  $\pi$ . We can find the longest

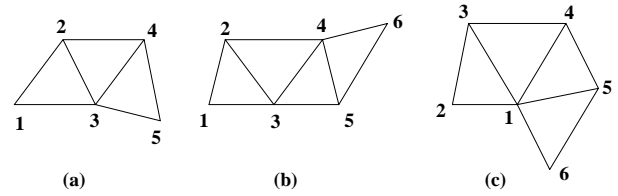


Figure 2: Hamiltonian triangulations of length 3 and 4.

prefix subpath,  $\pi'$ , of  $\pi$  for which the corresponding set of triangles is sequential; this is readily done in time  $O(|\pi'|)$ , in the same manner that one tests for a triangulation being sequential ([2]). By Lemma 1, we know that  $|\pi'| \geq 3$ . Thus, using a simple greedy algorithm, in which we iteratively select the longest prefix of  $\pi$  that is sequential, and then remove it, we see that the result is a set of at most  $\lceil n/3 \rceil$  sequential tristrips that partition  $S$ . (This bound is tight: consider a fan triangulation.) Similarly, if we allow both sequential and fan tristrips, we will need at most  $\lceil n/4 \rceil$  tristrips (and one can again show that the bound is tight). The result is the following lemma:

**Lemma 3** *Using only (pure) sequential tristrips, a Hamiltonian triangulation has encoding cost at most  $n + 2\lceil n/3 \rceil \leq (5n + 4)/3$ . Using both sequential and fan tristrips, the encoding cost is at most  $n + 2\lceil n/4 \rceil \leq (3n + 3)/2$ . Such encodings can be computed in  $O(n)$  time.*

Now consider an arbitrary connected triangulation,  $S$ , with each edge belonging to at most two triangles. Let  $\mathcal{T}$  denote a spanning tree of the adjacency graph  $\mathcal{G}$  for  $S$ . Then, since each node of  $\mathcal{G}$  has degree at most three, we know that we can root  $\mathcal{T}$  in such a way that it is a (rooted) binary tree, simply by rooting it at any node of degree at most two.

**Lemma 4** *Any rooted binary tree can be partitioned in linear time into a set of paths, such that every path in the partition, except for at most one, consists of at least three nodes.*

*Proof.* We give a constructive proof, based on the following

**Path Peeling Algorithm:**

- (1) Let  $v$  be a node of maximum depth among nodes that have two children in the tree. If no such  $v$  exists, go to step (3).  
Then, the subtree rooted at  $v$  must consist of a *path*,  $\pi_v$ ; otherwise, there exists a node of greater depth than  $v$  that has two children, contradicting the choice of  $v$ . Further,  $\pi_v$  has at least three nodes.
- (2) Remove  $\pi_v$  from the tree. If the tree is empty now, stop; otherwise, go to step (1).
- (3) (No node of the tree has two children.) In this case, the tree is already a path (possibly consisting of only one or two nodes), so we report it and stop.

The algorithm can be implemented in linear time, since each node can be labeled with its depth and degree and each path that is peeled off can be charged to the nodes of the path.  $\square$

**Theorem 5** *For any connected triangulation  $S$  of  $n$  triangles, one can compute in  $O(n)$  time a pure sequential encoding of  $S$  with at most  $2n + 1$  vertices. In the case of sequential strips allowing zero-area triangles (resp., sequential and fan strips), one obtains a bound of  $\lceil (7/4)n \rceil + 1$  (resp.,  $1.8n + 1.2$ ).*

*Proof.* We let  $\mathcal{T}$  be any spanning tree (e.g., a depth-first search tree, which is computable in linear time) for the dual graph  $\mathcal{G}$  of  $S$ . We then apply the “Path Peeling Algorithm” of Lemma 4 to decompose  $\mathcal{T}$  into paths each of length 3 or more (except, possibly, for one path). Now consider one such path  $\pi$ , and let  $i$  be the number of its triangles. Obviously, the costs are 3, 4, and 5 for paths of length  $i = 1, 2$ , and 3. By Lemma 3 we get a cost of  $(5i + 4)/3 \leq 2i$  for  $i \geq 4$ . Summing over all the paths yields the claim. (Note that one path may be of length  $i = 1$ , requiring  $2i + 1$  vertices.)

A similar analysis can be applied to the case in which we allow both sequential and fan tristrips, or in which we allow zero-area triangles in the sequential strips.  $\square$

### Corollary 6

$$\begin{aligned} (5/3)n &\leq \mathcal{C}_s(n) \leq 2n + 1. \\ (5/3)n &\leq \mathcal{C}'_s(n) \leq \lceil (7/4)n \rceil + 1. \\ \mathcal{C}_{s,f}(n) &\leq 1.8n + 1.2. \end{aligned}$$

*Proof.* The upper bounds come directly from the theorem above. The lower bounds come from a family of examples, one of which is shown in Fig. 3. The dual tree for the triangulation can be partitioned into  $n/4$  paths each having four nodes, as shown in the middle figure; this is the result produced by our Path Peeling Algorithm. One can argue that the optimal partitioning, however, is obtained by breaking the tree into pairs of paths consisting of a 5-node path and a singleton, as shown on the right.  $\square$

The above corollary says that the worst-case number of vertices per triangle is between  $5/3$  and 2 for pure sequential strips, between  $5/3$  and  $7/4$  for sequential strips allowing zero-area triangles, and at most 1.8 for mixed strips. In Section 6, we see that, in practice, our heuristics do better than these worst-case bounds, with our best method averaging about 1.21 vertices per triangle (allowing zero-area triangles in sequential strips).

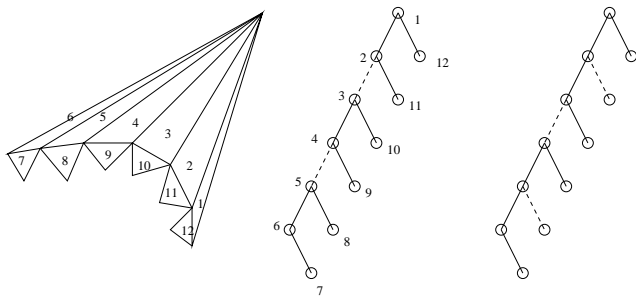


Figure 3: Lower-bound example for the Corollary.

## 4 Stripification Algorithm

Our stripification algorithm consists of five steps:

- (1) computing a triangulation of faces of the model that are not already triangles;
- (2) constructing a spanning tree,  $\mathcal{T}$ , in the dual graph  $\mathcal{G}$  of the triangulation;
- (3) partitioning  $\mathcal{T}$  into a set of paths, corresponding to Hamiltonian triangulations;
- (4) greedily decomposing the corresponding Hamiltonian strips into sequential or fan tristrips; and

- (5) concatenating short tristrips into longer tristrips, using a set of postprocessing heuristics applied to the result of Step (4); see Section 5.

For (1), we have integrated and enhanced the FIST [10] system, which robustly triangulates polygonal models, even if highly degenerate or corrupted. Our modification of FIST enables us to output triangulations of convex faces that are pure tri-strips, or to leave convex faces untriangulated (and triangulate only all concave faces).

For Step (2), we have implemented standard depth-first search (DFS) and breadth-first search (BFS), and we have also devised a *hybrid* variant that does DFS, but returns to the *highest* level node not yet fully explored.

For (3), we apply our “bottom up” Path Peeling Algorithm, from Lemma 4 of the last section. This guarantees that each path (Hamiltonian strip), except possibly for one, will have at least three triangles. We also devised an *optimal* algorithm, based on dynamic programming, to partition  $\mathcal{T}$  into a minimum number of sequential tristrips; see Section 5.

The overall goal in Step (2) is to build a spanning tree  $\mathcal{T}$  of  $\mathcal{G}$  that has a small number of nodes of degree two, as this will result in a small number of paths generated in Step (3) of the algorithm. BFS tends to generate trees that resemble balanced binary trees. Therefore, the number of nodes in a BFS-tree that have two children may be large. DFS and the hybrid search both tend to generate more degree-one nodes than BFS, thus reducing the number of paths generated in Step (3).

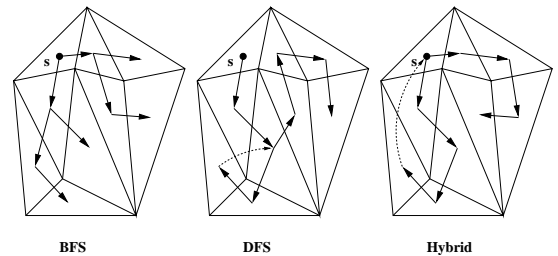


Figure 4: Three graph searching algorithms.

There is a choice of which triangle to visit next during the construction of  $\mathcal{T}$  whenever the graph search has entered a triangle with two unvisited neighbors. Obviously, one can simply perform *random marching* and pick the next triangle randomly. One heuristic for picking a “good” next triangle has been implemented by SGI in *tomesh*: this heuristic picks the triangle with the least number of unvisited neighbors. Using this heuristic in Fig. 4(b) will find one single path that covers  $S$ . Another heuristic is to keep track of the side on which the current triangle has been entered from the previous triangle, and to choose the next triangle such that one alternates the turn. This *alternate-turn marching* can be expected to help to get longer and therefore fewer sequential strips.

We implement and test all three graph search algorithms, BFS, DFS, and hybrid search. For each search algorithm, we generate the following three types of strips:

- (1) sequential strips only,
- (2) fan strips only,
- (3) both sequential and fan strips.

If both sequential and fan strips are generated, we favor sequential strips. That is, a fan strip will start only if the greedy decomposition in Step (4) encounters four consecutive triangles that cannot be encoded with a sequential strip.

Our algorithm runs in overall *linear* time in the worst case, after triangulation (Step (1)). In practice, FIST has been shown to take only linear time for Step (1), see [10], especially since most faces

tend to be triangles, quadrilaterals, or low-cardinality polygons. (In theory, triangulation can be done in worst-case linear time for faces without holes, and in  $O(n \log n)$  in general.)

## 5 Improved Stripification Algorithm

### Taking Care of Orientations

In computing tristrips, we take care to respect the *orientation* of the leading triangle in each tristrip, as this gives the renderer crucial information about the sign (orientation) of the normal vector. Following the convention that normal vectors point to the outside of the object, this means that the vertices of the first triangle have to appear in counter-clockwise (CCW) orientation when viewed from the outside.

Taking care of the orientation of the first triangle is no problem for a tristrip of odd length, as we can simply specify its vertices in reverse order. However, it creates a problem for tristrips of even length, if their original vertex order is inappropriate. For instance, the first triangle of the tristrip in Fig. 2(a) is oriented CW, but specifying the vertices in reverse order, (5,4,3,2,1), changes the orientation of the first triangle to CCW. However, a reversal of the vertex order does not cure the problem for the tristrip in Fig. 2(b). One may either break it up or add “swap” vertices to make it a strip of odd length, depending on whether or not swaps are permitted.

### Dynamic Programming Optimization

The goal of our Dynamic Programming (DP) algorithm is to minimize the number of sequential strips needed to cover a given spanning tree  $\mathcal{T}$ . We engineered our algorithm to ensure correct orientations of the tristrips, provided that the input model has consistent orientations for all the triangular faces.

For each node  $v \in \mathcal{T}$ , we define the objective function  $f(v, i)$  to be the minimum number of sequential strips that can be derived from the subtree rooted at  $v$ , in “mode  $i$ ”, where “mode  $i$ ” has the following meanings, for  $i = 0, 1, 2, 3, 4$ :

- 0: no sequential strip enters  $v$ ;
- 1: a strip enters  $v$  with a left turn and an even parity;
- 2: a strip enters  $v$  with a right turn and an even parity;
- 3: a strip enters  $v$  with a left turn and an odd parity;
- 4: a strip enters  $v$  with a right turn and an odd parity.

Here, we say a triangle is *entered with a left turn* by strip  $s$  if its vertex encoding in  $s$  complies with its orientation. Otherwise, we say it is *entered with a right turn*. We also say that a triangle is entered by strip  $s$  with an even parity if it is an even-numbered triangle within  $s$ ; otherwise, it is entered with an odd parity. Note that  $f(v, i)$  does not include in its count the strip (if any) that enters node  $v$ .

Assume that  $v$  has two children, “left” and “right”, which are denoted by  $l$  and  $r$ . Then we can establish the following recursive relations. (We note that this also works for models that do not have consistent orientations.) For  $i = 0$ :

$$f(v, 0) = \min\{1 + f(l, 0) + f(r, 0), \\ 1 + \min\{f(l, 1), f(l, 2)\} + f(r, 0), \\ 1 + \min\{f(r, 1), f(r, 2)\} + f(l, 0), \\ 1 + (f(l, 3) \text{ or } f(l, 4)) + (f(r, 3) \text{ or } f(r, 4))\}.$$

$f(v, 0)$  is computed by optimizing over all possible cases:

1. the node consists of a singleton strip;
2. a strip starts from  $v$  and enters its left child with a left (resp., right) turn and an even parity;
3. a strip starts from  $v$  and enters its right child with a left (resp., right) turn and an even parity;
4. a strip passes through  $v$ , and enters its left child with either a

left or right turn and its right child with either a left or right turn, both with an odd parity.

Similarly, we can establish the following recursive relations. For  $i = 1$  and  $i = 2$ :

$$f(v, i) = \min\{f(l, 0) + f(r, 0), \\ f(l, 3) \text{ or } f(l, 4) + f(r, 0), \text{ or} \\ f(r, 3) \text{ or } f(r, 4) + f(l, 0)\}.$$

For  $i = 3$  and  $i = 4$ :

$$f(v, i) = \min\{f(l, 0) + f(r, 0), \\ f(l, 1) \text{ or } f(l, 2) + f(r, 0), \text{ or} \\ f(r, 1) \text{ or } f(r, 2) + f(l, 0)\}.$$

If  $f(v, 1) = f(l, 0) + f(r, 0)$ , i.e., if the optimal value is achieved when the strip stops at  $v$  with an even parity and with a left turn, then its orientation could not be corrected. Thus, in this case we set  $f(v, 1)$  to  $\infty$ .

For a leaf node  $v$ , the boundary conditions are  $f(v, 0) = 1$ ,  $f(v, 1) = \infty$ ,  $f(v, 2) = f(v, 3) = f(v, 4) = 0$ .

The optimal value is given by  $f(v_r, 0)$ , where  $v_r$  is the root of  $\mathcal{T}$ . To actually build sequential strips, one has to store information about the optimum decomposition at every node.

The computation can be done in linear time, by traversing the tree in a bottom-up fashion. There are only a constant number of cases per node and each node is visited exactly once.

**Theorem 7** *In  $O(n)$  time, one can compute an optimal decomposition of a tree into a minimum number of pure sequential tristrips.*

### Strip Concatenation

In this section, we discuss how to concatenate tristrips in Step (5) of the Stripification Algorithm in order to reduce both the number of vertices rendered and the number of tristrips. We start with explaining how to concatenate sequential tristrips if both zero-area and duplicate triangles are allowed. (The simple modifications for using only zero-area triangles are omitted here.)

Let  $\sigma_1 = (v_1, v_2, v_3, \dots, v_n, v_{n+1}, v_{n+2})$  and  $\sigma_2 = (u_1, u_2, u_3, \dots, u_m, u_{m+1}, u_{m+2})$  be two sequential tristrips in the triangulation  $S$ . Assume that either the first or last triangle of  $\sigma_1$  is a neighbor of the first or last triangle of  $\sigma_2$ . We will explain strip concatenation for the case that  $(v_n, v_{n+1}, v_{n+2})$  and  $(u_1, u_2, u_3)$  are neighbors.

The result of strip concatenation should be that three consecutive vertices in the new strip specify either a triangle in  $S$  or a newly introduced degenerate triangle. We use “+” to denote the binary operation that combines two sequential strips. We have to consider three cases.

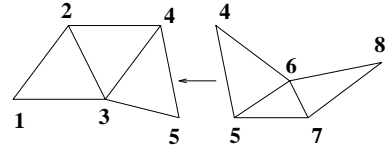


Figure 5: Concatenation that reduces the encoding cost by two.

If  $v_{n+1} = u_1$  and  $v_{n+2} = u_2$  then

$$\sigma_1 + \sigma_2 \leftarrow (v_1, \dots, v_n, u_1, u_2, u_3, \dots, u_{m+2}).$$

This first case corresponds to only one configuration, as illustrated in Fig. 5. We have

$$(1, 2, 3, 4, 5) + (4, 5, 6, 7, 8) \leftarrow (1, 2, 3, 4, 5, 6, 7, 8).$$

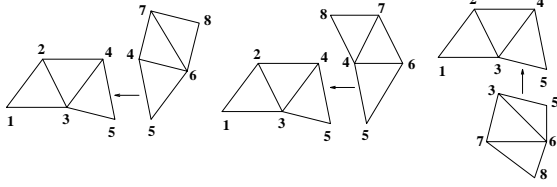


Figure 6: Concatenation that reduces the encoding cost by one.

If  $v_n = u_2$  or  $v_{n+1} = u_2$  or  $v_{n+1} = u_3$ , and  $v_{n+2} = u_1$  then

$$\sigma_1 + \sigma_2 \leftarrow (v_1, \dots, v_n, v_{n+1}, u_1, u_2, u_3, \dots, u_{m+2}).$$

This second case breaks down into three configurations, as illustrated in Fig. 6. The concatenations corresponding to the three configurations are

$$\begin{aligned} (1, 2, 3, 4, 5) + (5, 4, 6, 7, 8) &\leftarrow (1, 2, 3, 4, 5, 4, 6, 7, 8), \\ (1, 2, 3, 4, 5) + (5, 6, 4, 7, 8) &\leftarrow (1, 2, 3, 4, 5, 6, 4, 7, 8), \\ (1, 2, 3, 4, 5) + (5, 3, 6, 7, 8) &\leftarrow (1, 2, 3, 4, 5, 3, 6, 7, 8). \end{aligned}$$

Else

$$\sigma_1 + \sigma_2 \leftarrow (v_1, \dots, v_n, v_{n+1}, v_{n+2}, u_1, u_2, u_3, \dots, u_{m+2}).$$

It can be easily checked that this third case corresponds to four other configurations, in which two strips can only be linked without achieving a reduction in the number of vertices.

The concatenation of two fan strips is simple. Similar to the discussion above, let  $\sigma_1 = (v_1, \dots, v_{n+1}, v_{n+2})$  and  $\sigma_2 = (u_1, u_2, u_3, \dots, u_{m+2})$  be two *fan* strips. Without loss of generality, we assume that the triangles  $(v_1, v_{n+1}, v_{n+2})$  and  $(u_1, u_2, u_3)$  are neighbors. Then  $\sigma_1$  and  $\sigma_2$  may be linked if and only if  $v_1 = u_1$  and  $v_{n+2} = u_2$ , and  $\sigma_1 + \sigma_2 = (v_1, \dots, v_{n+1}, v_{n+2}, u_3, \dots, u_{m+2})$ .

Tristrips of length one (i.e., single triangles) are special. The vertex sequence in the strip  $\sigma = (v_1, v_2, v_3)$  can be permuted in cyclic order. This introduces variations in the basic algorithms for strip concatenation.

Our concatenation algorithm achieves only the optimal concatenation for one pair of neighboring strips  $\sigma_1$  and  $\sigma_2$ . However, a strip of length one has three neighboring strips which may be candidates for concatenation, and a strip of length greater than one has two terminal triangles, i.e., two candidates for concatenation. Thus, the order of concatenation may matter. For a better global optimization, three concatenation stages are performed, each of which constructs concatenations that reduce the number of vertices by 2, 1, or 0 vertices per concatenation. A direct addressing table can be used to store, for each triangle, the index of its incidental strip  $\sigma$ . One pass through the table checks each triangle that has a valid strip index. Each such triangle's neighboring triangles are looked up in the table to find their incidental strips, which are neighbors of  $\sigma$ . One such strip is chosen to be linked to  $\sigma$  to reduce the most number of vertices. The global optimization as mentioned above may be achieved by multiple passes through the table. A hash table may be used to replace the direct addressing table to save memory and computing time, though our implementation chooses the latter to ease the coding complication and yet the memory consumption and the increase in running time is negligible compared to the other parts of the program.

In theory, two tristrips can potentially be concatenated in Step (5) by using either a zero-area triangle or a duplicate triangle. In practice, rendering triangles repeatedly may cause visual artifacts, depending on how the graphics hardware treats duplicates. With our stripification code the user can specify whether to allow no “swap” vertices, zero-area triangles, or zero-area triangles and duplicate triangles.

## 6 Results

We have conducted experiments on various platforms; here we report only on one representative platform: a Sun Ultra 30 with 512MB memory, running Solaris 2.6. The CPU-time consumption of our code, and of the other codes, was obtained by using the C system function “getrusage()”. We report both the system and the user time. Of course, any file I/O and similar pre- or postprocessing is not included in the timings reported. All CPU times are given in milliseconds. All codes were compiled with GNU’s gcc, using the optimization level “-O2”.

We compared FTSG against the leading publicly available system, STRIPE 2.0. (STRIPE 2.0 has been released recently; it is orders of magnitude faster than the published [8] version, STRIPE 1.0.) All subsequent results have been obtained using STRIPE 2.0. For comparison purposes we also tested SGI’s tomesh.

Making STRIPE report the same type of statistics data as FTSG and tomesh turned out to be a cumbersome task. STRIPE has most I/O operations tightly interwoven with its algorithm, and its accounting is not entirely reliable. We modified STRIPE in order to exclude file I/O from timing. Also, we decided to parse the output generated by STRIPE in order to obtain the statistics data reported here. (Of course, the reliability of STRIPE was tested by running the original code.)

Our datasets included many standard models available on the web, with sizes ranging from 32 vertices to 543,652 vertices (1,087,716 triangles). We tested a variety of 107 models, including models of buildings and crafts (designed on CAD systems), and models of animals and fictional characters that were typically derived from scanned data. We also included a few highly regular polyhedral models of machining tools. Typically, those models required only one or a very small number of sequential strips.

We first tried to determine experimentally which combinations of heuristics yielded the best results. Parameters in our tests included

- searching the dual graph by means of depth-first (-dfs), breadth-first (-bfs), or hybrid search (-hyb);
- using random marching (-rnd) or alternate-turn marching (-alt) during the graph search;
- enabling the use of zero-area triangles (-zero) or duplicate triangles (-dup) during the strip concatenation.
- generating sequential strips (-seq), fan strips (-fan), or both sequential and fan strips (-seq -fan).

In all our tests FTSG was required to generate tristrips that are consistent with the orientation of the faces of a model (if such a consistent orientation existed).

The results of our tests are summarized in Table 1, which lists, averaged over all our models, the average numbers of vertices per triangle and the average CPU time (in milliseconds) per triangle. The different heuristics are arranged in sorted order according to their performance.

Using the alternate-turn heuristic for depth-first search and allowing zero-area triangles for concatenating purely sequential tristrips yielded the best results, with an average of 1.23 vertices per triangle. (The use of duplicate triangles further decreases this number by about 1%, but duplicate triangles have been ruled out since they might cause visual artifacts on some graphics hardware.) In general, depth-first search with the alternate-turn heuristic yielded slightly better results than the hybrid search, and both performed by far better than breadth-first search.

It is interesting to see that using fan strips in conjunction with sequential strips does not help to decrease the vertex count. This observation is also confirmed by Fig. 7. It shows the percentage

method				avg. V/T	CPU (ms)
-dfs	-alt	-seq	-zero	1.23	0.009
-dfs	-alt	-seq	-fan	1.24	0.010
-hyb	-alt	-seq	-zero	1.25	0.013
-hyb	-alt	-seq	-fan	1.27	0.013
-hyb	-rnd	-seq	-zero	1.28	0.012
-dfs	-alt	-seq		1.29	0.009
-dfs	-rnd	-seq	-zero	1.29	0.011
-hyb	-alt	-seq		1.32	0.012
-hyb	-alt	-seq	-fan	1.33	0.011
-bfs	-alt	-seq	-zero	1.33	0.010
-hyb	-rnd	-seq		1.36	0.010
-dfs	-rnd	-seq		1.38	0.009
-dfs	-alt	-fan		1.57	0.010
-hyb	-alt	-fan		1.58	0.014
-hyb	-alt	-fan	-zero	1.58	0.013

Table 1: Performance of different heuristics implemented in FTSG: average vertices per triangle (“V/T”) and CPU time (in ms).

of the models the second-to-fourth best heuristics produced results that were at least  $k$  times the results achieved by using “-dfs -alt -seq -zero”, or at most  $k$  times the results for “-dfs -alt -seq -zero” (for  $k < 1$ ). The combination “-dfs -alt -seq -fan -zero” is nearly identical (but slightly worse) than “-dfs -alt -seq -zero”. We conclude that using fan strips does not seem to pay off.

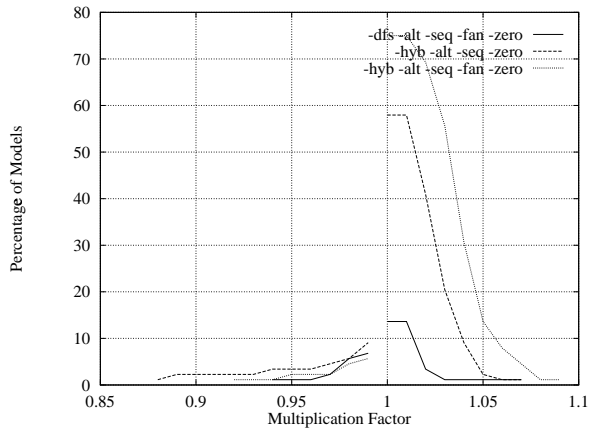


Figure 7: Encoding quality of heuristics of FTSG relative to using “-dfs -alt -seq -zero”

We also determined the best combination of heuristics for the DP algorithm. The results of our tests are summarized in Table 2. As above, depth-first search using alternate-turn marching yielded the best result: on average, it needs 1.21 vertices per triangle (if concatenation by means of zero-area triangles is allowed), and consumes about 0.014 milliseconds per triangle. When using pure sequential strips the average vertex count goes up to 1.24.

We compared FTSG against STRIPE 2.0. Since STRIPE cannot handle models with non-convex polygonal faces, which amounted to about 50% of our models, we used FIST to generate triangulations of our models. (STRIPE crashes or produces garbage when applied to concave faces.) In one series of tests, all convex faces were left untriangulated in order to allow STRIPE to triangulate them according to its own heuristics. STRIPE turned out to be unreliable at times, and we had to restrict our tests to

method				avg. V/T	CPU (ms)
-dfs	-alt	-DP	-zero	1.21	0.014
-hyb	-alt	-DP	-zero	1.22	0.016
-dfs	-alt	-DP		1.24	0.013
-hyb	-rnd	-DP	-zero	1.25	0.016
-hyb	-alt	-DP		1.26	0.015
-dfs	-rnd	-DP	-zero	1.26	0.014
-hyb	-rnd	-DP		1.30	0.015
-dfs	-rnd	-DP		1.31	0.013

Table 2: Performance of different heuristics for the DP algorithm implemented in FTSG: average vertices per triangle (“V/T”) and CPU time (in ms).

method				avg. V/T	CPU (ms)
-dfs	-alt	-DP	-zero	1.21	0.014
-dfs	-alt	-seq	-zero	1.23	0.009
-dfs	-alt	-DP		1.24	0.013
tomesht				1.36	0.027
STRIPE (convex faces)				1.36	0.263
STRIPE (fully triangulated)				1.39	0.071

Table 3: Performance of the different stripification codes: average vertices per triangle (“V/T”) and CPU time (in ms).

those 88 (out of 107) models that STRIPE could handle. (FTSG has no problem handling any form of polyhedral model as it always uses FIST as a preprocessing tool for triangulating polyhedral faces. Note too that STRIPE becomes more reliable when using FIST as a front end.) For comparison purposes we also tested SGI’s tomesht.

The results are summarized in Table 3, and in Fig. 8 and Fig. 9. (We used FTSG’s heuristics “-dfs -alt -DP -zero” for these comparisons.) The plots show the percentage of the models in which the other two codes produced results that were at least  $k$  times the results of FTSG, or at most  $k$  times the results for FTSG (for  $k < 1$ ).

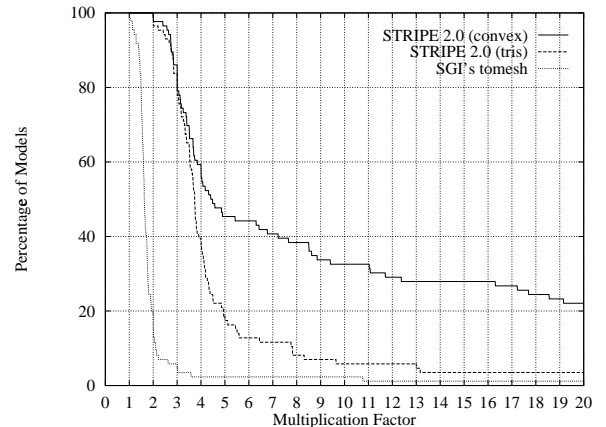


Figure 8: CPU-time comparison of the stripification codes w.r.t. FTSG “-dfs -alt -DP -zero”

FTSG was always faster than STRIPE or tomesht. On average, FTSG needs 0.014 milliseconds (ms) per input triangle, tomesht needs 0.027ms, and STRIPE needs 0.071ms for fully triangulated faces respectively 0.263ms for non-triangulated convex faces. For

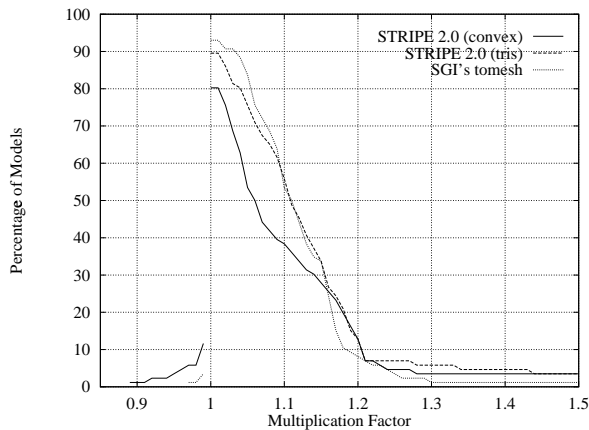


Figure 9: Encoding quality of the stripification codes w.r.t. FTSG “-dfs -alt -DP -zero”

20% of the models, `tomes` needed at least twice as much CPU time as FTSG. STRIFE needed at least twice as much CPU time as FTSG for all models. For fully triangulated models it was at least five times as slow as FTSG for 18% of the models, and at least 10 times as slow for 6% of the models. For models with convex faces left untriangulated, it was at least five times as slow as FTSG for 45% of the models, 10 times as slow for 33%, and 20 times as slow for 22% of the models.

On average, FTSG needs 1.21 vertices per input triangle, `tomes` needs 1.36 vertices, and STRIFE needs 1.39 vertices for fully triangulated faces and 1.36 vertices for non-triangulated convex faces.<sup>1</sup> For 80% of the models FTSG performed better than STRIFE (if convex faces are left untriangulated). For fully triangulated models this percentage goes up to 93% for `tomes` and 89% for STRIFE. For 38% respectively 55% of the models STRIFE needed at least 10% more vertices than FTSG. (For `tomes`, this percentage was 53%.) While `tomes` hardly ever generated encodings with fewer vertices than FTSG, STRIFE was better for 11% of the models, if convex faces are left untriangulated. As we have seen, this win for 11% of our models comes at the price of a CPU consumption that is drastically worse than FTSG’s CPU consumption. Note that FTSG performs better than STRIFE or `tomes` even when it does not use swap vertices, see the entry for “-dfs -alt -DP” in Table 3.

STRIFE generates significantly fewer strips than FTSG, but still uses more vertices. We note that the objective of the current DP algorithm is to minimize the number of sequential tristrrips. Sometimes this comes at the expense of quite a few singleton strips (i.e., strips of length one). We have started to experiment with modified objective functions that allow us to minimize the number of strips while avoiding the generation of too many singleton strips. We expect that a judiciously modified objective function will decrease the number of vertices per triangle.

FTSG shows no significant variation of the main performance parameters. Its cpu-time consumption (per triangle) varies very little, as does the number of vertices per triangle. Summarizing, FTSG is orders of magnitude faster than the published STRIFE 1.0,

<sup>1</sup>There are some cases in which our results for STRIFE differ from what is reported in [8]. There are two reasons for this: (1). STRIFE does not handle concave polygonal faces (thus, we pre-triangulated all concave faces prior to applying STRIFE); and (2). there are some errors and inconsistencies in how STRIFE counts vertices. Thus, our results are based on a careful parsing of the *output* set of tristrrips from STRIFE (run with its default options).

and performs much more predictably (and typically better) than STRIFE 2.0 and `tomes`, while being substantially faster on average.

## 7 Conclusion

We have shown that our methods, implemented in FTSG, compare favorably with prior methods of stripification, both in terms of speed and in terms of quality. We anticipate that an improved objective function for the DP approach will help to improve our results even further. We also plan to optimize over all spanning trees, e.g., by branch and bound or by randomization. This will give the optimum (minimum) encoding of a model, and will allow us to compare our heuristics to the optimum.

**Acknowledgments.** We thank Michael Deering, Henry Sowitzal, and Karel Zikan of Sun Microsystems for their input on this research; Sun also provided a grant and equipment in support of this work. We are also partially supported by Boeing, Bridgeport Machines, NSF (CCR-9504192,CDA-9626370,CCR-9732220), and a Fulbright award (J. Mitchell). We thank Estie Arkin and Steve Skiena for several useful discussions on this research.

## References

- [1] K. Akeley, P. Haeberli, and D. Burns. `tomes.c`: C Program on SGI Developer’s Toolbox CD, 1990.
- [2] E. M. Arkin, M. Held, J. S. B. Mitchell, and S. S. Skiena. Hamiltonian Triangulations For Fast Rendering. *Visual Comput.*, 12(9):429–444, 1996.
- [3] R. Bar-Yehuda and C. Gotsman. Time/Space Tradeoffs For Polygon Mesh Rendering. *ACM Trans. Graph.*, 15(2):141–152, 1996.
- [4] M. M. Chow. Optimized Geometry Compression For Real-time Rendering. In *IEEE Visualization ’97 Proceedings*, pages 347–354, San Francisco, CA, 1997. ACM/SIGGRAPH Press.
- [5] M. Deering. Geometry Compression. *Comput. Graph.*, pages 13–20, 1995. Proc. SIGGRAPH ’95.
- [6] M. Denny and C. Sohler. Encoding A Triangulation As A Permutation Of Its Point Set. In *Proc. 9th Canad. Conf. Comput. Geom.*, pages 39–43, 1997.
- [7] F. Evans. STRIFE: <http://www.cs.sunysb.edu/~stripe/>, 1998.
- [8] F. Evans, S. S. Skiena, and A. Varshney. Optimizing Triangle Strips For Fast Rendering. In *IEEE Visualization ’96 Proceedings*, pages 319–326, Oct. 1996.
- [9] S. Gumhold and W. Straßer. Real Time Compression Of Triangle Mesh Connectivity. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 133–140, Orlando, FL, July 1998.
- [10] M. Held. Efficient And Reliable Triangulation Of Polygons. In *Proc. Comput. Graphics Internat.*, pages 633–643, 1998.
- [11] B. Speckmann and J. Snoeyink. Easy Triangle Strips For TIN Terrain Models. In *Proc. 9th Canad. Conf. Comput. Geom.*, pages 239–244, 1997.
- [12] G. Taubin, A. Guéziec, W. Horn, and F. Lazarus. Progressive Forest Split Compression. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 123–132, Orlando, FL, July 1998.
- [13] G. Taubin and J. Rossignac. Geometric Compression Through Topological Surgery. *ACM Trans. Graph.*, 17: (2) 84–115 APR 1998.
- [14] C. Touma and C. Gotsman. Triangle Mesh Compression. In *Proc. 24th Conf. Graphics Interface*, pages 26–34, San Francisco, CA, 1998. Morgan Kaufmann Publishers.

Figure 10: Results of FTSG stripification of the cow. Above: Stripified cow, with different strips having different colors; Below: A zoomed view, with spanning tree edges highlighted on the cow's head.

Figure 11: Results of FTSG stripification of the shark. Above: Stripified shark, with different strips having different colors; Below: A zoomed view, with spanning tree edges highlighted.