An Iterative Stripification Algorithm Based on Dual Graph Operations

Massimiliano B. Porcu, and Riccardo Scateni

Dipartimento di Matematica e Informatica, University of Cagliari, Cagliari, Italy

Abstract

This paper describes the preliminary results obtained using an iterative method for generating a set of triangle strips from a mesh of triangles. The algorithm uses a simple topological operation on the dual graph of the mesh, to generate an initial stripification and iteratively rearrange and decrease the number of strips. Our method is a major improvement of a proposed one originally devised for both static and continuous level-of-detail (CLOD) meshes and retains this feature. The usage of a dynamical identification strategy for the strips allows us to drastically reduce the length of the searching paths in the graph needed for the rearrangement and produce loop-free triangle strips without any further controls and post-processing.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling – Geometric algorithms, languages, and systems

1. Introduction

A triangle strip is a set of connected triangles where a new vertex implicitly defines a new triangle. Triangle strips are used to accelerate the rendering of objects represented as triangle meshes, in a pre-processing stage the mesh is partitioned in a set of triangle strips (possibly composed of one isolated triangle) and then each strip is passed to the Graphics Processing Unit (GPU) for rendering. The advantage of the strip representation over rendering each triangle separately, is that it makes it possible to reduce the number of vertices sent to the GPU from *3n* (where *n* is the number of triangles in the mesh) to *n+2* in the best case.

Given the current advances in computer architecture, resulting in having the CPU-GPU communication as the most common bottleneck of the whole visualization process, it appears evident that a good stripification strategy could virtually improve by a factor of three the CPU-GPU bandwidth (at the best case, when a single strip, representing the whole mesh, is produced) and, consequently, the whole visualization.

Unfortunately it has been proven [8, 1] that a problem equivalent to searching the optimal single strip (finding a Hamiltonian path on the dual graph) is an NP-complete problem, thus the stripification process should be based on local heuristics.

We decided to follow an approach consisting of operating on the dual graph of the mesh (as described in details in sections 3 and 4) using a single topology operator, previously referred to as a *tunnelling operator*[16]. Since we can apply the operator to a general dual graph we show that we can use our algorithm in two ways: to optimize an existing stripification or to generate a new stripification from scratch. The implementation of the algorithm relies on a single relevant parameter, the *tunnel length*, which influences both the time spent to stripify the mesh and the final set of strips obtained (number and mean length). Thus is very easy to use even for non expert users.

As in the original formulation we retain the property to apply the algorithm to Continuous Level of Detail meshes (CLOD) making it possible to repair the stripification when adding triangles in case of progressive transmission of the mesh[10].

The rest of this work is organized as follows: in section 2 we briefly go over the previous work done in geometry compression, focusing on stripification; we then show, in section 3, the relations existing between the triangle mesh and its dual graph and introduce the tunnelling operator; section 4 is dedicated to point out the differences between our implementation and the original proposed by Stewart; in section 5 we show the preliminary results obtained using our algorithm on several meshes widely used in literature for benchmarking; and, finally, in section 6 we draw our conclusions and describe the future evolutions of this work.

2. Previous Work

The term *geometry compression* applies to two separated but largely overlapping tasks:

- Reducing the data (triangle mesh) size to be sent over the network;
- Changing the way in which the triangle mesh is described when sent from the CPU to the GPU.

2.1. Geometry compression

Deering[3] in 1995 was the first to introduce the term *geometry compression*, to describe a set of techniques capable of reducing the space occupancy of a *generalized triangle mesh* statistically encoding XYZ positions, RGB colors and normals. These techniques operate mainly on the *geometry* of the mesh (i.e., the positions and the attributes of the vertices) relying on the triangle mesh structure to compress the information on the *topology* (i.e., how the vertices are connected to form the triangles).

Subsequent works[10, 17, 18, 9], instead, centered their attention on the problem of compressing the description of the topology arriving at a relevant result with the Edgebreaker method[13, 14] proposed by Rossignac in 1999 which uses less than two bits per triangle to encode a planar mesh homeomorphic to a disc.

All these techniques need a decompression stage that is not yet implementable in commercial graphics hardware, even using new programmable boards. This means that they are very efficient for transmission and archiving but cannot be used for feeding the GPU.

2.2. Stripification Techniques

Rearranging the order in which the vertices are stored is another way to face the problem. The strips obtained are smaller than the original mesh when coming to the final rendering since, while the single triangle needs 3 vertices for its visualization to be sent to the GPU, the triangle strip needs *n+2* vertices to be sent to the GPU to render *n* triangles. The optimal single strip encoding the whole mesh would reduce the number of vertices sent to the GPU by a factor of three.

The great advantage of using triangle strips consists of the availability of such a primitive in the OpenGL graphics library. Generating a stripification of a mesh means to be able to feed the GPU with the obtained structure without any further effort. It is actually to point out that OpenGL supports, without any vertex replication, only the sequential triangle strips. Generalized strips could thus bring to send more than once some vertices to the GPU. It is beyond the scope of our current implementation to tackle this problem, but we plan to investigate this.

This explains why a lot of effort has been spent in elaborating good heuristics to stripify a mesh[7, 2, 15, 19, 5, 4, 12, 6].

It is worthwhile to explicitly mention a technique[11] that uses a greedy algorithm to take advantage of the caching strategy of the graphics boards, thus differentiating in a way from the others cited, and the work[16] that first proposed to use the tunnelling operator on the dual graph, which is described in details in the next section.

3. The Triangle Mesh and its Dual Graph

Each triangle mesh can be alternatively represented by its *dual graph*. It is a graph in which each node is associated to a triangle of the original mesh and an edge represents an adjacency relation. One trivial property of such a graph is that each node has, at most, three incident arcs. In case the original mesh is homeomorphic to a sphere and has genus 1, each node has exactly three incident arcs (see figure 1).

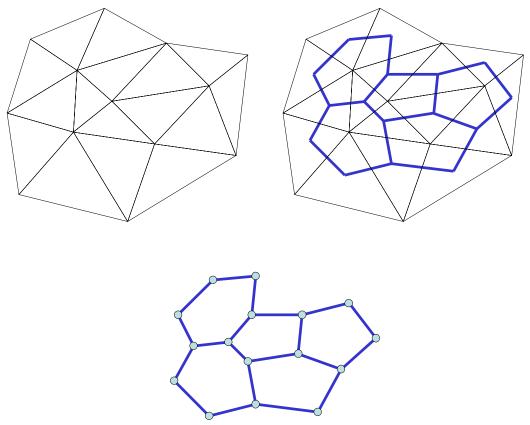
Figure 1: *A triangle mesh and its dual graph.*

3.1. The Tunnelling operator

The tunnelling algorithm, as proposed by Stewart[16], performs the stripification of the mesh using a simple topological operation on its dual graph.

To do so we need to *color* all graph edges in two possible ways (see figure 2):

solid edges linking nodes associated to triangles in the same strip;

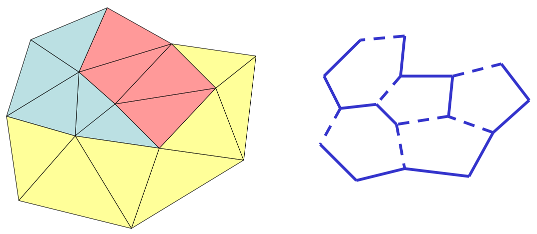

Figure 2: *A stripified mesh (each color encodes a different strip) and its dual graph.*

dashed edges linking nodes associated to adjacent triangles not belonging to the same strip.

In every node there are, at most, two incident solid edges. The nodes with only one incident solid edge are *terminal nodes* (corresponding to terminal triangles of the stripification). The nodes with three incident dashed nodes correspond to isolated triangles in the stripification.

The first step of the operation consists, then, of searching a special kind of path in the graph called *tunnel*. A tunnel is an alternating sequence of solid and dashed edges, starting and ending with a dashed edge, connecting two terminal nodes. Its length is always odd and we denote by *k*-tunnel a tunnel of length *k*.

If a tunnel is found, the second step consists, simply, of complementing the path, that is, changing each solid edge in a dashed edge and vice-versa. After this operation the number of solid paths (strips in the triangulation) on the graph is reduced by one. See figure 3 for example.

It is quite easy to understand how this technique can be used both to improve an existing stripification or to create a stripification from scratch. In the latter case the starting dual graph will have only dashed edges and every path of length one can be chosen as a tunnel. It is worthwhile to point out that isolated triangles are always considered as terminal nodes of a one-triangle strip.

The main problem when implementing the algorithm is the possibility that the graph traversal for tunnelling could select paths that, when complemented, would generate loops. It is thus necessary to follow two additional rules (we can call them the *no-loop rules*) during the tunnel search to avoid this situation:

1. The last edge in a tunnel cannot connect two nodes belonging to the same strip (see figure 4).
2. When a non-final dashed edge, *e* say, in the tunnel joins two nodes belonging to the same strip, the next solid edge should go back in the direction of the leading node of *e* (see figure 5).

To be able to respect the no-loop rules, it is necessary to

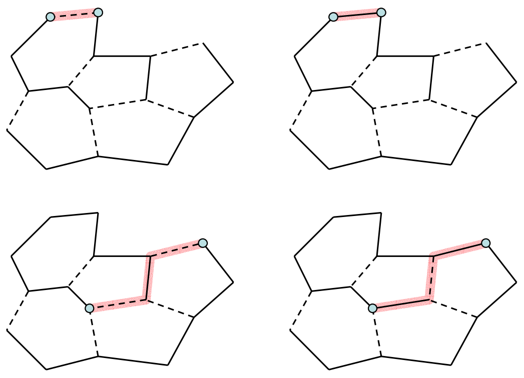

Figure 3: *An example of tunnelling. In the top row a 1-tunnel is found; in the bottom row there are no 1-tunnels but only a 3-tunnel. Notice that the number of strips decreases from three to two after the first operation and to one after the second.*

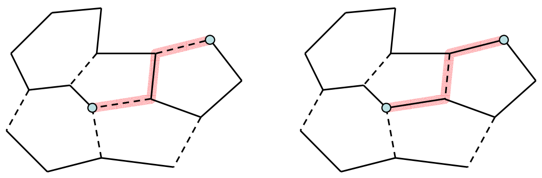

Figure 4: *An incorrect tunnelling that generates a loop.*

distinguish between the different strips in the graph. This is done tagging each node of the graph (triangle) with an identifier corresponding to the strips it belongs to.

3.2. Tunnelling on CLOD

As stated by Stewart[16] a great advantage of the tunnelling technique over other stripification techniques is that it can used to repair triangulations that are damaged by changes in mesh topology, such as occur in CLOD meshes. Even if it is not yet possible with the present version of our algorithm to apply the tunnelling operator to progressive meshes, we regard to this as the first feature to add in the future.

4. Our approach

It should be clear, by now, how a stripification via the tunnelling operator is performed on a generic triangle mesh:

1. Generate the dual graph of the triangulation;
2. Perform a breadth-first search, starting from a randomly chosen terminal node, finding the shortest valid tunnel (i.e., a tunnel not violating the no-loop rules) if one exists;

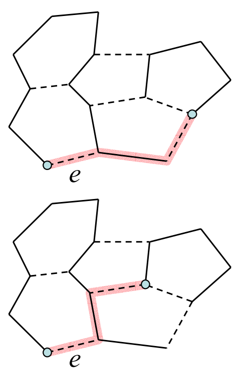

Figure 5: *The non-final edge* e *in the tunnel joins two nodes belonging to the same strip. Of the two next possible steps, we must select the one corresponding to the direction that comes back to the leading node of* e *(bottom row), otherwise it will generate a loop (top row). One such step always exists because the leading and trailing nodes of* e *are in the same strip.*

3. Complement the edges of the tunnel;
4. Iterate the process until no more tunnels are found.

To prevent loops, to each strip in the graph one associates a unique identifier, and each node in the graph is tagged with the identifier of the strip it belongs to. The isolated triangles are all tagged with different identifiers. When complementing the tunnel edges, the strips identifiers change and we should define a strategy to update the node identifiers accordingly. Recall that a node identifier changes when a dashed edge switches to a solid edge; the node id changes taking the identifier of the previous node in the tunnel. Moreover, the update should propagate to the other nodes belonging to the same strip (see figure 6).

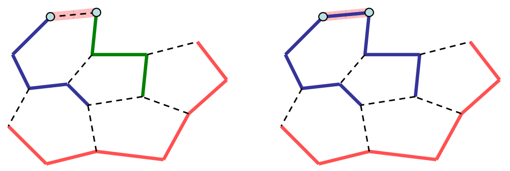

Figure 6: *Identifier propagation along a strip. The identifier is color coded.*

To avoid a connection between nodes belonging to the same strip (a loop), it is necessary to check that these nodes do not have the same identifier. In the original tunnelling algorithm[16] the node's identifier was checked only at the beginning of the search (using a *static identifier* strategy); instead, it is essential to update the identifiers of nodes at each step of the tunnel construction (using a *dynamic identifier* strategy). Only in this way the algorithm is robust and the no-loop rules are enough to prevent the creation of any loop (see figure 7).

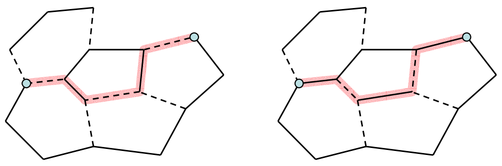

Figure 7: *An example of tunnel producing a loop even respecting the no-loop rules. In this case, our dynamic id propagation strategy prevents from generating a loop.*

To explain how we implemented the dynamic identifier strategy, we need to introduce a new definition.

There are two kinds of nodes traversed during the tunnel construction:

primary nodes Nodes that are part of the tunnel;
secondary nodes Nodes whose identifier is updated, because they belong to the same strip of a primary node.

In each tunnel search it can happen that a node is traversed more than once. To prevent loop creation one should consider that a primary node cannot be visited, as *primary*, more then once. Instead, primary and secondary nodes can be visited, as *secondaries*, an arbitrary number of times.

The original proposed mechanism to prevent triangles loop creation was to follow very restrictive rules about primary and secondary nodes. Doing so no dynamical id update is required if a node can be visited only once, both as primary or secondary node. This choice considerably reduces the algorithm and data structure complexity. On the other hand, this limits strongly the search and forces us to discard many in-principle valid paths. As a consequence, as evident in the next section, the whole stripification process is limited, and the time efficiency is poor because the breadth-first search depth is too long.

4.1. The Data Structure

Our implementation of the tunnelling algorithm is not trivial, since it requires a supporting data structure able to keep track of the dynamical identifier update. Moreover, since the data structure should support an *undo* operation, we should, in fact, be able to discard any change, if the tunnel search do not end successfully.

We chose to implement the dual graph using a double linked random access triangle list. Each element of the list, a graph node, stores at most three links to each adjacent triangle. The status of each link (solid or dashed) is represented by a logical flag. To support the dynamical id strategy, the flag is already changed during the tunnel search but it is possible to backtrack the change in case the search does not end with a valid tunnel.

We use a more complex structure for node identifiers. We use a flag to mark a node as visited as primary (it can happen only once). Since a node can be visited, as secondary, an arbitrary number of times, a single temporary identifier is not enough. Instead, we have a stack of identifiers in each node, to manage identifier changes with standard push (update the node identifier) and pop (discard the last update) operations.

5. Results and Discussion

The tunnelling algorithm behavior depends mainly on two parameters. One parameter, explicitly set by the user, is the tunnel length, the other, chosen by the algorithm, is the set of starting nodes in the graph.

By increasing the tunnel length, we trade off an improvement of the quality of the stripification for an increase in computing time. Our results show that the tunnelling algorithm is able to produce a stripification with a small number of strips even when it starts from a mesh with several hundreds thousands of triangles, if we use long enough tunnels. The length, anyway, is quite small (we never exceeded sixty), compared with the results reported by Stewart (up to tunnel of one thousand steps).

5.1. Stripification Quality

In figure 8 we show the number of stripes vs maximum tunnel length obtained stripifying four different models very often used for benchmarking (details in the figure caption). In all the cases, the algorithm has been used to generate stripe from scratch. To take in account the variations given by the random starting nodes selection, we performed ten different runs with different starting points. The values reported in the graphics correspond to mean values. For a more quantitative judgement, we also report the values in table 1.

In table 2 we compare the number of strips obtained, respectively, by our algorithm, the original tunnelling algorithm and the SGI algorithm on the Stanford bunny and dragon models. We compare the results only for these two models because they are the only ones for which we have homogeneous data available. It is evident the improvement obtained with the dynamic identifier strategy.

5.2. Time Performance

We compared the tunnelling algorithm time performances with the original tunnelling algorithm and a stripification method based upon the SGI algorithm. The comparison is not totally accurate since it is impossible to drive the algorithm to produce exactly the same number of strips, but the differences are irrelevant.

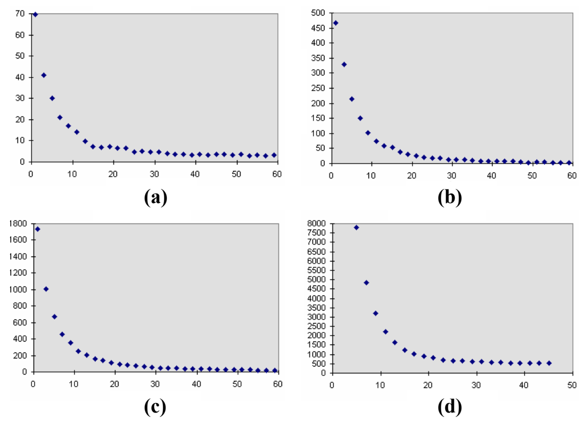

Figure 8: *Number of stripes vs maximum tunnel length for (a) NVidia simplified bunny (1,999 tris), (b) UNC oil-pump (20,544 tris), (c) Stanford bunny (69,451 tris) and (d) Stanford dragon (871,414 tris). All the benchmarks done on a PC with a Pentium IV 1.7 GHz CPU, with 256 MB of RAM.*

Table 1: *Results of the stripification of the models listed in figure 8 at different maximum tunnel length. The stripification of the dragon gives memory faults when increasing the maximum tunnel length over 45.*

Tunnel Length	NVidia Bunny	Oil Pump	Stanford Bunny	Stanford Dragon
1	70	466	1,733	23,900
5	30	213	671	7,805
9	17	102	353	3,211
13	10	59	203	1,622
17	7	38	142	1,040
21	7	25	96	808
25	5	18	75	673
29	5	14	60	610
33	4	13	44	577
37	4	9	37	554
41	4	7	36	541
45	3	7	29	517
49	3	4	27	-
53	3	5	25	-
57	3	4	22	-

The comparison with the original tunnelling algorithm shows (results are reported in table 3) how our algorithm becomes much more efficient on large datasets. We limited our stripification quality to match the results reported. Notice that the maximum tunnel length used in this case is not the maximum overall.

The comparison with the SGI algorithm is not obvious. Our results are largely better then SGI in term of number of strips and mean length. We cannot, thus, apply the two meth-

Table 2: *Number of strips obtained by the SGI algorithm, the original tunnelling algorithm and our algorithm from the Stanford bunny and the Stanford dragon.*

Model	SGI algorithm	Original algorithm	Our algorithm
Bunny	648	158	22
Dragon	17,401	1,798	517

Table 3: *Time in seconds to obtain almost the same number of strips by the original tunnelling algorithm and our algorithm on the Stanford bunny (158 strips, maximum tunnel length=17) and the Stanford dragon (1,798 strips, maximum tunnel length=13). Times are scaled to take in account different CPU speed.*

Model	Original algorithm	Our algorithm
Bunny	17.40	23.33
Dragon	13,500.00	699.08

ods at the same data set and simply compare the execution time, because the final results are somehow different: using the tunnelling method we obtain much less strips in more time. Using the tunnelling method to reproduce the results obtained with the SGI one, that is a stripification with about the same number of stripes, we tried to compare execution time with the same final result (without any considerations about stripe quality). In this case, as we can see in table 4, the times are of the same order of magnitude.

6. Conclusions and Future Work

We described a stripification algorithm based on a simple topological operation on the dual graph of the triangle mesh that is robust and easy to use. Only one parameter is needed to drive its execution. This is a major performance and quality improvement over a similar algorithm proposed. The preliminary results obtained make us confident that we shall be able to implement a version of the algorithm capable to operate also on CLOD meshes. It could be used to repair the inconsistencies in the stripification of a LOD when inserting new triangles.

The choice of the search seeds is still an open issue. We plan to elaborate on strategies different from the current ones that choose randomly the starting node and move at random in the graph. One goal of such a strategy should also be the generation of strips being as much sequential as possible, to accommodate the current requirements of the graphics libraries.

We also plan to investigate the limits of the stripification algorithm when applied to huge meshes, eventually adopting an out-of-core scheme allowing the stripification of meshes of any size.

Table 4: *Time in seconds to obtain almost the same number of strips by the SGI algorithm and our algorithm on the Stanford bunny (705 strips, maximum tunnel length=3), and the Stanford dragon (17,653 strips, maximum tunnel length=3).*

Model	SGI algorithm	Our algorithm
Bunny	0.57	1.94
Dragon	7.08	22.52

Acknowledgements

We like to thank Riccardo Piergallini for fruitful discussions during the writing of this paper.

References

1. ARKIN, E. M., HELD, M., MITCHELL, J. S. B., AND SKIENA, S. S. Hamiltonian triangulations for fast rendering. *The Visual Computer 12*, 9 (1996), 429–444. 1
2. CHOW, M. M. Optimized geometry compression for real-time rendering. In *IEEE Visualization '97* (Nov. 1997), pp. 346–354. 2
3. DEERING, M. F. Geometry compression. In *Proceedings of SIGGRAPH 95* (Aug. 1995), Computer Graphics Proceedings, Annual Conference Series, pp. 13–20. 2
4. EL-SANA, J., EVANS, F., KALAIAH, A., VARSHNEY, A., SKIENA, S., AND AZANLI, E. Efficiently computing and updating triangle strips for real-time rendering. *Computer-Aided Design 32*, 13 (Oct. 2000), 753–772. 2
5. EL-SANA, J. A., AZANLI, E., AND VARSHNEY, A. Skip strips: Maintaining triangle strips for view-dependent rendering. In *IEEE Visualization '99* (Oct. 1999), pp. 131–138. 2
6. ESTKOWSKI, R., MITCHELL, J. S. B., AND XIANG, X. Optimal decomposition of polygonal models into triangle strips. In *Proceedings of the eighteenth annual symposium on Computational geometry* (2002), ACM Press, pp. 254–263. 2
7. EVANS, F., SKIENA, S. S., AND VARSHNEY, A. Optimizing triangle strips for fast rendering. In *IEEE Visualization '96* (Oct. 1996), pp. 319–326. 2
8. GAREY, M. R., JOHNSON, D. S., AND TARJAN, R. E. The planar hamiltonian circuit problem is np-complete.

SIAM Journal of Computing 5, 4 (Dec 1976), 704–714. 1

9. GUMHOLD, S., AND STRASSER, W. Real time compression of triangle mesh connectivity. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), ACM Press, pp. 133–140. 2

10. HOPPE, H. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), ACM Press, pp. 99–108. 1, 2

11. HOPPE, H. Optimization of mesh locality for transparent vertex caching. In *Proceedings of SIGGRAPH 99* (Aug. 1999), Computer Graphics Proceedings, Annual Conference Series, pp. 269–276. 2

12. ISENBURG, M. Triangle strip compression. *Computer Graphics Forum 20*, 2 (2001), 91–101. 2

13. ROSSIGNAC, J. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 5* (Jan. 1999), 47–61. 2

14. ROSSIGNAC, J., AND SZYMCZAK, A. Wrap&zip decompression of the connectivity of triangle meshes compressed with edgebreaker. *Computational Geometry 14*, 119–135. 2

15. SPECKMANN, B., AND SNOEYINK., J. Easy triangle strips for TIN terrain models. In *Canadian Conference on Computational Geometry* (1997), pp. 239–244. 2

16. STEWART, A. J. Tunneling for triangle strips in continuous level-of-detail meshes. In *Graphics Interface* (June 2001), pp. 91–100. 1, 2, 3, 4

17. TAUBIN, G., AND ROSSIGNAC, J. Geometric compression through topological surgery. *ACM Transactions on Graphics 17*, 2 (Apr. 1998), 84–115. 2

18. TOUMA, C., AND GOTSMAN, C. Triangle mesh compression. In *Graphics Interface '98* (June 1998), pp. 26–34. 2

19. XIANG, X., HELD, M., AND MITCHELL, J. S. B. Fast and effective stripification of polygonal surface models. In *1999 ACM Symposium on Interactive 3D Graphics* (Apr. 1999), pp. 71–78. 2