Tunneling for Triangle Strips in Continuous Level–of–Detail Meshes

A. James Stewart

Dynamic Graphics Project
Department of Computer Science
University of Toronto

Abstract

This paper describes a method of building and maintaining a good set of triangle strips for both static and continuous level–of–detail (CLOD) meshes. For static meshes, the strips are better than those computed by the classic SGI and STRIPE algorithms. For CLOD meshes, the strips are maintained incrementally as the mesh topology changes. The incremental changes are fast and the number of strips is kept very small.

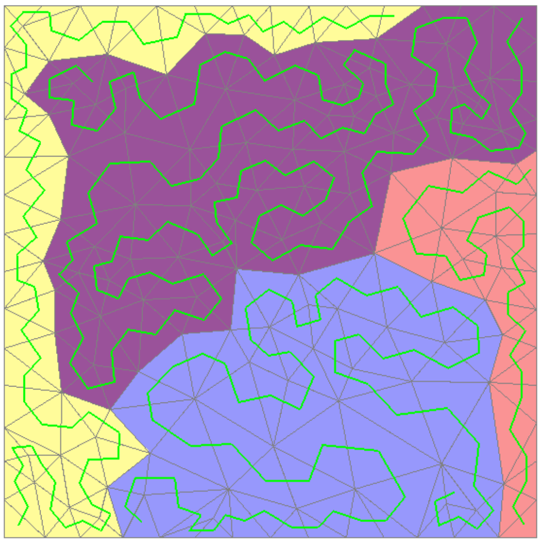
1 Introduction

A *triangle strip* is a sequence of triangles in which adjacent triangles share an edge and nonadjacent triangles are interior–disjoint. Triangle strips are used to accelerate the rendering of objects represented as triangle meshes: The mesh is decomposed into a set of triangle strips, and each triangle strip is rendered independently.

A strip of k triangles is sent to the graphics processor as a sequence of $k+2$ vertices: three vertices for the first triangle and one for each additional triangle. The goal of a *stripification algorithm* is to compute a small set of triangle strips whose union covers the mesh. Fewer, longer strips permit fewer vertices to be sent to the graphics processor. This is quite important with current video cards, where CPU–to–card bandwidth can be a limiting factor in rendering speed.

This paper presents an algorithm to maintain a good stripification of a triangle mesh whose topology changes over time. Such meshes are used in many applications, most commonly to control the level of detail in order to balance rendering quality with rendering speed: Areas of low perceptual importance are represented with a few large triangles, while many small triangles are used in areas of high perceptual importance. As the viewpoint moves, the mesh changes topology.

Changes in the topology of a mesh require changes in its stripification: A triangle inserted into the mesh should be included in some triangle strip, to prevent it from forming its own "singleton strip." A triangle removed from the mesh might be removed from the middle


4 strips for 270 triangles

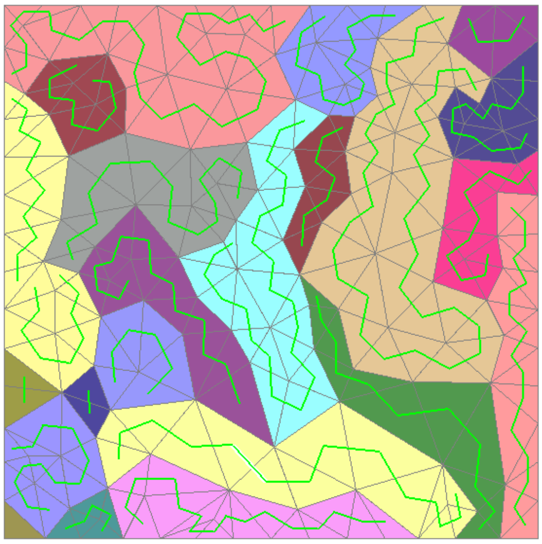

22 strips for 270 triangles

Figure 1: Good and bad stripifications

of a strip, breaking that one strip into two. Repeated removals and insertions can cause the number of strips to increase dramatically, effectively negating the advantage of the original stripification.

This paper describes a "tunneling operator" whose application to a stripification can reduce the number of strips. The operator does so by building a "tunnel" through the mesh in order to join the endpoints of two different triangle strips, thus reducing the total number of strips by one (more on this in Section 3). This operator is useful in several ways:

- For **static meshes**, repeated tunneling reduces the number of triangle strips, optimizing a stripification.
- **Continuous level–of–detail (CLOD) meshes** [11, 16] do not require topological transformations to be performed in a specific order, so it is not practical to precompute stripifications of all possible topologies. Instead, the tunneling operator can be applied on–the–fly in order to repair any damage caused by the mesh transformations, thus maintaining a high–quality stripification.
- With **progressive meshes** [10], a mesh is encoded as a small, reduced mesh plus a sequence of "vertex split" operations which, when performed in order on the reduced mesh, reproduce the original mesh. Each intermediate mesh is a reduced–detail version of the original mesh.

  In a similar manner, a stripification of the original mesh can be encoded as a stripification of the reduced mesh plus a precomputed sequence of tunneling operations which, when performed in parallel with the vertex splits, reproduce the original stripification. Each intermediate stripification is of high quality, allowing the intermediate mesh to be efficiently rendered.

The tunneling operator is quite simple to implement and does not require much more than triangle adjacency information. It can easily be tuned to trade execution time for stripification quality, which is useful for real–time applications that use CLOD meshes. Experiments have shown that static mesh stripifications produced by tunneling are of high quality, and that this high quality stripification is maintained even in CLOD meshes.

The tunneling operator builds good *generalized strips* in which there is no constraint on the structure of the strip. *Sequential strips* are more restrictive: A triangle strip enters each of its triangles at an "entry edge" and leaves that triangle on the left or the right of the two remaining "exit edges:" With sequential strips, the exit edge must alternate between left and right with each successive triangle. In the OpenGL rendering model, generalized strips require an extra vertex per triangle when such a constraint is broken; no extra vertex is necessary in the SGI GL model. Given the strong recent interest in improved video card architectures (in particular, programmable strip construction [15], generalized triangle meshes [4], and transparent vertex caching [12]) it will likely not be long before generalized strips have the same rendering cost as sequential strips in the more popular rendering models.

2 Related Work

2.1 Stripifications of Static Meshes

A Hamiltonian path in a triangle mesh is a sequence of pairwise adjacent faces in which each face is visited exactly once. Such a sequence is an optimal stripification. But testing whether a given triangle mesh admits a Hamiltonian path is NP–complete [2], so researchers have resorted to heuristics to build good stripifications: The classic SGI algorithm [1] greedily constructs a strip by extending it incrementally to the next adjacent node of lowest valence, which tends to avoid short strips. The widely used STRIPE algorithm [7] finds large areas of quadrilaterals, triangulates them and stripifies them, and greedily adds the remaining triangles. Other methods [21, 17] construct a spanning tree of the dual graph and traverse it to form triangle strips.

The tunneling algorithm of this paper is also a heuristic approach which can build a stripification or can improve upon a given stripification. It can typically reduce the number of triangle strips produced by the SGI and STRIPE algorithms (from 705 and 917 strips, respectively, to 158 strips for the 69,000–triangle Stanford bunny, for example).

2.2 Stripifications in CLOD Meshes

The real advantage of tunneling is that it can incrementally repair triangulations that are damaged by changes in mesh topology, such as occur in continuous level–of–detail meshes. In CLOD meshes, the mesh topology varies with the viewpoint in order to concentrate detail in perceptually important areas, and to remove detail (for faster rendering) in unimportant areas. As the viewpoint moves, the mesh topology is changed with "edge collapse" and "vertex split" operations (described in Section 3.1). The variation is called continuous because the mesh topologies differ by a single edge between sufficiently close viewpoints.

The Skip Strip algorithm [6] maintains a stripification of one type of CLOD mesh: the view–dependent progressive mesh (VDPM). The VDPM and its highest–resolution stripification are transmitted to the client, which then builds a skip list structure on top of the VDPM vertex hierarchy. For any topology of the mesh, the skip list structure permits an efficient reconstruction of the unbroken subsequences of the highest–resolution strips. But where a highest–resolution strip is broken (e.g. where the faces it traverses are not in the current mesh) there appears to be no provision to repair the break.

In contrast, the tunneling algorithm acts to repair any broken strips, thus maintaining a high quality stripification. It also does not need much extra memory, requiring little more than the strip and triangle adjacency information for the current mesh.

Other work has concentrated on stripification of *hierarchical* CLOD meshes. Such meshes have a very constrained structure which allows a good stripification to be defined for all configurations of the mesh. The tunneling approach of this paper cannot compete with most of the hierarchical methods, since it does not rely upon any knowledge of the structure of the mesh.

A small sampling of hierarchical approaches is mentioned here: For CLOD terrains represented with a quadtree, Lindstrom *et al* [14] have an elegant recursive algorithm for building generalized triangle strips. The ROAM renderer [5] uses a fast incremental approach to maintain *sequential* strips of four or five triangles each, on average. Recursive space filling curves are used by Velho *et al* [20] to build generalized triangle strips.

2.3 Transmission of Stripifications

There has been much recent work on the compression of triangle mesh connectivity [8, 18, 19], but it appears that only one author has considered encoding a stripification along with the connectivity: Isenburg [13] has described such a method which uses only one or two additional bits per triangle over the pure connectivity encoding methods.

As with the other connectivity methods, Isenburg's goal is not to provide a progressive mesh transmission, but rather to minimize the total transmission cost. Our tunneling approach has a much larger transmission cost when used with progressive meshes. But, on the other hand, it does allow intermediate meshes to be rendered with high quality triangulations.

2.4 Cache–Optimizing Stripifications

Deering introduced *generalized triangle meshes* [4] which use a vertex cache of more than two vertices to decrease vertex transfers from the CPU to the graphics card. The idea is to exploit those vertices that are currently in the cache in order to minimize the number of times a given vertex is sent to the graphics card. Bar–Yehuda and Gotsman [3] have shown that a cache of size $\mathcal{O}(\sqrt{n}\ )$ is necessary to minimize vertex transmission in a mesh of size n. Hoppe [12] has described heuristics to construct triangle strips that are optimized for a given cache size.

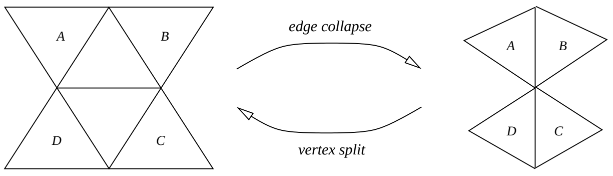


Figure 2: An edge collapse removes an edge and two faces. A vertex split inserts an edge and two faces.

The tunneling algorithm does not consider cache coherence of triangle strips: Its only goal is to maintain a minimum–size stripification. But tunneling can be thought of as one rule in a grammar of graph transformations, and it might be possible (in future work) to add more rules to the grammar to take into account cache coherence and sequential — rather than generalized — stripifications.

3 Triangle Strip Tunneling

The paper will discuss the tunneling algorithm in the context of progressive meshes (PMs) and view–dependent progressive meshes (VDPMs). The tunneling algorithm is not limited to such meshes: It can be used with any CLOD mesh, such as those formed by vertex removal [16], for example.

3.1 Local Repairs Are Not Enough

Progressive meshes perform two operations to modify a mesh: edge collapses and vertex splits. Shown in Figure 2, an edge collapse removes an edge and its two adjacent faces from the mesh, while a vertex split does the inverse to insert an edge and two faces into the mesh.

An edge collapse can cause a triangle strip to be broken into two, and a vertex split can create a new triangle strip of length two (see Figure 3). While these problems do not occur with all collapses or splits, they occur in sufficient number to quickly fragment a stripification, as will be shown in Section 4.

In certain configurations (such as those in the top and middle rows of Figure 3) there is no local modification of the stripification that can reduce the number of strips. It is clear that a more global modification is sometimes required.

3.2 Tunneling

In the **dual graph** of a triangulation, there is a node corresponding to each mesh triangle and there is an edge joining each pair of nodes whose corresponding mesh triangles are adjacent (see Figure 4). Note that each non–boundary node has three adjacent graph edges. There are two types of graph edges: **Strip edges** join nodes whose corresponding triangles are adjacent on the same strip; all others are **nonstrip edges**.

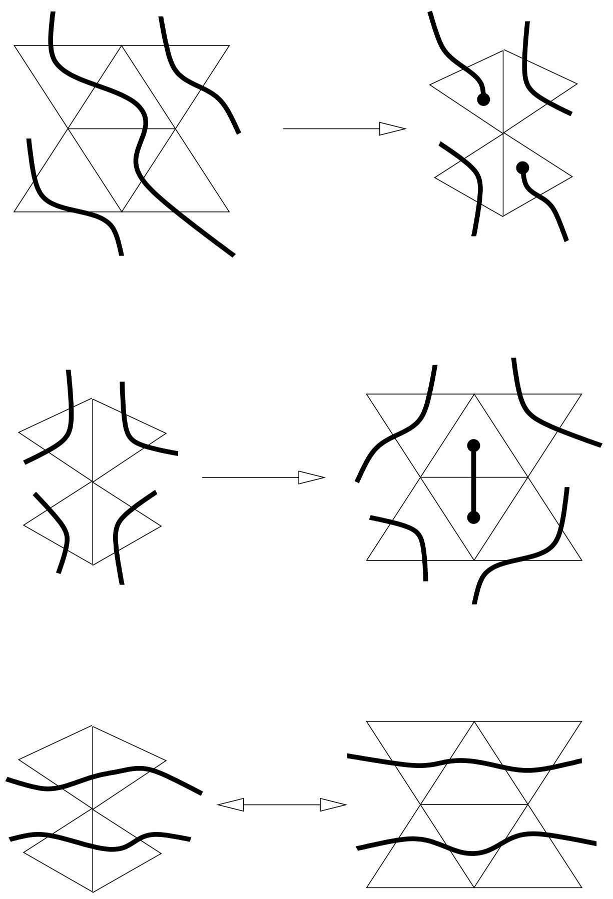


Figure 3: (top) An edge collapse can cause a strip that crosses the collaped edge to be split into two. (middle) A vertex split can create a new, isolated strip which joins the two new triangles. (bottom) Some collapses and splits do not cause any damage to the stripification.

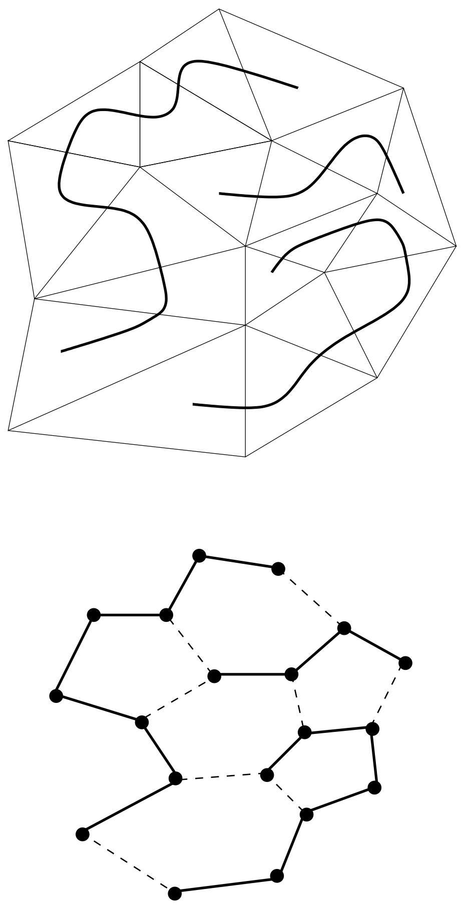


Figure 4: (top) A triangle mesh with three strips. (bottom) The dual graph with two types of edges: solid strip edges and dashed nonstrip edges.

A **tunnel** is an alternating sequence of strip and nonstrip graph edges, which starts and ends with nonstrip edges and which connects two nodes that each have fewer than two adjacent strip edges (i.e. nodes whose corresponding triangles are at the ends of their strips). If a tunnel exists in the dual graph, the number of strips can be reduced by one strip by complementing the status of each edge on the tunnel: Strip edges become nonstrip edges and vice versa.

Consider, for example, the graph fragment on the top of Figure 5 which contains four triangle strips. Two of the strips terminate at nodes A and B, between which a tunnel is shown. On the bottom of Figure 5 the edges of the tunnel are complemented, resulting in a stripification of only three strips.

The tunneling algorithm is conceptually very simple: Start at a source node on the end of a strip. Perform a breadth–first search in the graph to find the shortest path of alternating edges to another node at the end of a strip. Complement the edges of the tunnel.

The tunneling algorithm is repeated (starting at another source node) as long as a tunnel can be found. When no more tunnels can be found, the number of strips has reached a local minimum. Since it is NP–hard to find an optimal stripification, one cannot know (without a huge amount of additional work) whether the number of strips has also reached a *global* minimum. In fact, the final stripification depends upon the sequence of source nodes, so different sequences of source nodes typically yield different local minima.

There are some special cases which must be considered when performing the breadth–first search from one source node:

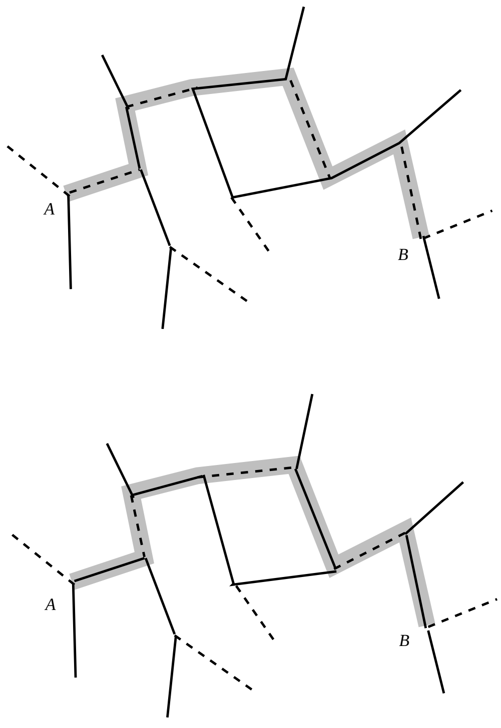


Figure 5: (top) A graph containing four triangle strips. A tunnel is highlighted between nodes A and B. (bottom) The same graph, but with the tunnel edges complemented. The graph now contains three triangle strips.

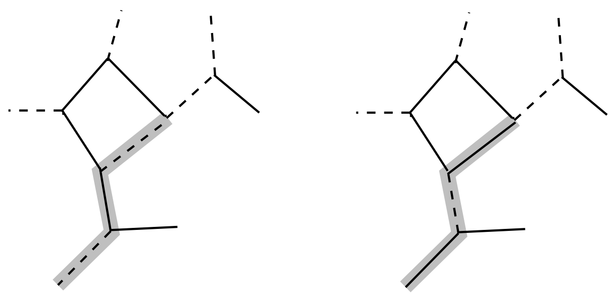


Figure 6: (left) A tunnel cannot end with an edge that joins two nodes on the same triangle strip. (right) Complementing such a tunnel introduces a triangle strip loop and does not reduce the total number of strips.

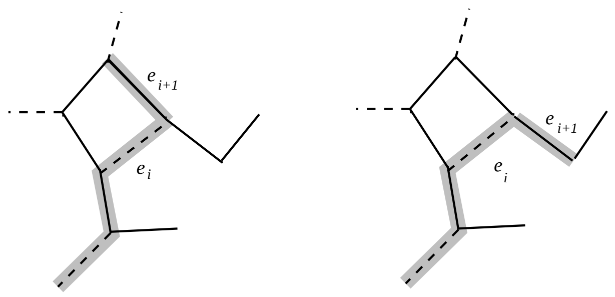


Figure 7: (left) A nonstrip edge e_i that joins nodes from the same triangle strip must be followed by a strip edge e_{i+1} that points toward the tail of e_i. (right) If e_{i+1} points away from the tail of e_i, a triangle strip loop is created upon complementing the edges.

- A tunnel consisting of a single nonstrip edge is possible. Complementing that edge simply joins two strips that end in adjacent triangles of the mesh.

- The tunnel can end at a node with zero adjacent strip edges, which corresponds in the mesh to an isolated triangle (i.e. a triangle on a strip containing only itself). Complementing the tunnel edges has the effect of adding the isolated triangle to a strip.

- The last (nonstrip) edge of the tunnel cannot connect two nodes belonging to the same strip (see Figure 6).

- Let each tunnel edge be directed outward from the source node at which the breadth–first search starts. If the i^{th} edge on a tunnel is a nonstrip edge that joins two nodes of the same triangle strip, the $i+1^{st}$ edge (if it exists) must turn in the direction toward the tail of the i^{th} edge (see Figure 7). Otherwise, a loop of triangles will be created upon complementing the i^{th} edge.

The special cases described above require that each node store the identifier of the triangle strip containing it. When strips are broken or joined by complementing the edges of the tunnel, all of the identifiers of these strips must be updated, along with any pointers (used by the renderer) to the ends of these triangle strips. This requires (in the worst case) a complete traversal of all the affected triangle strips.[1] This is the expensive part of the algorithm, but the experiments of Section 4 show that it is not prohibitively expensive.

The tunneling operation bears a remarkable resemblance to the classic "shortest augmenting path" technique for the Maximum–Weight Bipartite Matching problem [9]. That technique repeatedly finds paths of alternating matching/nonmatching edges in a bipartite graph and complements their status to increase the num-

[1] Up–trees may be used to join the triangle strips in (essentially) constant time, but up–trees are not useful for breaking the triangle strips.

ber of matching edges. The idea of an augmenting path is used in a variety of graph algorithms.

4 Applications of Tunneling

The tunneling operator can be used in several contexts: with static meshes to build or improve a stripification; with progressive meshes to encode a multiresolution stripification along with the PM; and with view–dependent progressive meshes to maintain a good stripification in the presence of an arbitrary sequence of edge collapses and vertex splits.

4.1 Static Meshes

For static meshes, simply start with a mesh of single–triangle strips and apply the tunneling operator repeatedly until no more tunnels can be found. As discussed in Section 3.2, this does not guarantee an optimal stripification (otherwise $P = NP$) but the experiments reported in Table 1 show that it consistently produces stripifications of high quality.

Experiments have shown that the speed of the tunneling algorithm can be increased with little effect on the stripification quality, simply by bounding the length of the tunnels. Each breadth–first search then runs in $\mathcal{O}(1)$ time (compared with $\mathcal{O}(n)$ time for tunnels of unbounded length). Figure 10 shows the tradeoff between tunnel length and stripification quality for the Stanford bunny. For the bunny, using a maximum tunnel length of 1000 took 198 seconds, while a maximum tunnel length of 75 took only 17.4 seconds.

4.2 View–Dependent Progressive Meshes

Good stripifications of view–dependent progressive meshes, and of CLOD meshes in general, can be maintained by a run–time application of the tunneling operator. After an edge split or vertex collapse, the tunneling operator is applied in two ways:

- Each of the four or six triangle faces in the immediate neighborhood of the collapse or split is checked. If such a face is at the end of a triangle strip, the tunneling algorithm is applied from that face. This step attempts to repair any damage caused by the collapse or split.

- In addition, a list is maintained of all mesh faces that lie on the end of a triangle strip. With each collapse and split, a small number of faces from that list are used as starting points for the tunneling algorithm (noted as “E 5” and “E 10” in Figure 11). This step attempts to reduce the number of strips elsewhere in the mesh. By only checking a few such faces with each collapse or split, the computational load is spread over time.

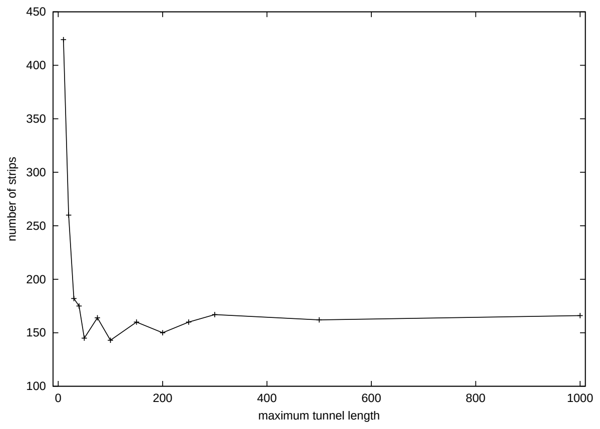


Figure 10: The number of strips produced for the Stanford bunny is shown for various maximum tunnel lengths. As the maximum tunnel length increases, the number of strips decreases as one would expect. There is some fluctuation because the number of strips is a local *minimum dependent upon the sequence of tunnels, which varies. From the graph, it appears that the tunnel length may be bounded at about 75 edges without affecting the quality of the stripification.*

Figure 11 compares variants of this tunneling algorithm to a non–tunneling approach which performs only *local repairs*: The local repairs join pairs of strips that both end at one of the four or six triangles involved in the collapse or split, or at a triangle immediately adjacent to one of those four or six triangles. The tunneling approach does much better than the local approach, and maintains a stripification of nearly–constant size.

The additional execution time required for tunneling is very slight: For the sequence of 5000 collapse and split operations reported on in Figure 11, the local–only algorithm took 59 seconds and the least aggressive “T 10, E 5” tunneling algorithm took 61 seconds.

It is interesting to note in Figure 11 that the number of strips increases as the maximum tunnel length decreases, and that the number of strips settles around some constant size which depends mainly upon the maximum tunnel length. It is possible that this “steady–state” phenomenon occurs when the maximum tunnel length and the triangle strip density come into equilibrium: If the maximum tunnel length is less than the average distance between strip endpoints, tunnels will not be found to join strips, and the number of strips will increase. Conversely, if the maximum tunnel length is greater than the average distance between strip endpoints, tunnels will be found and the number of strips will decrease. At some point this process reaches a state of equilibrium.

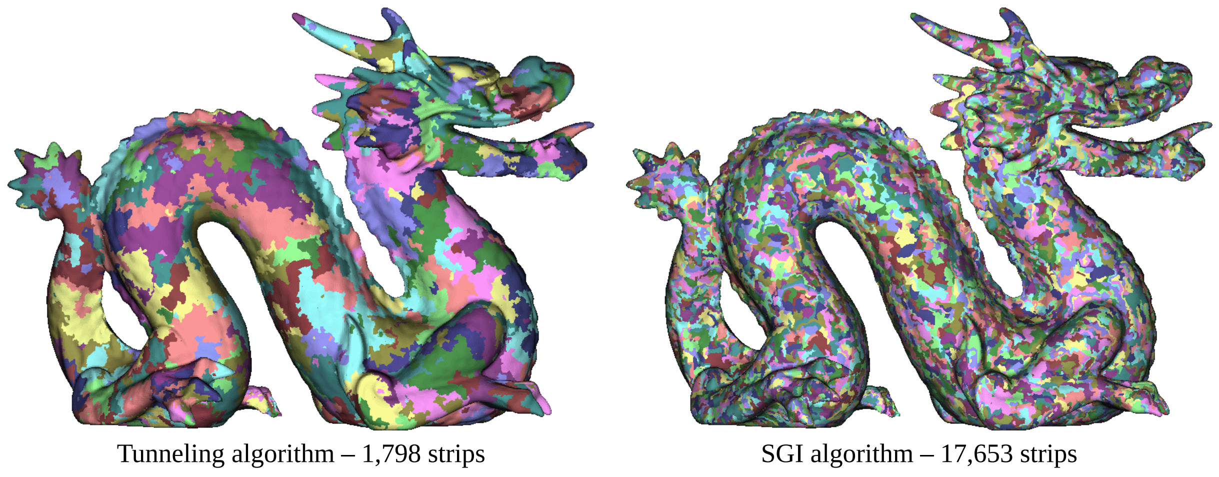

Tunneling algorithm – 1,798 strips
SGI algorithm – 17,653 strips

Figure 8: The Stanford dragon consisting of 871,414 faces, with stripifications by the tunneling and SGI algorithms. Each coloured area is covered by one strip.

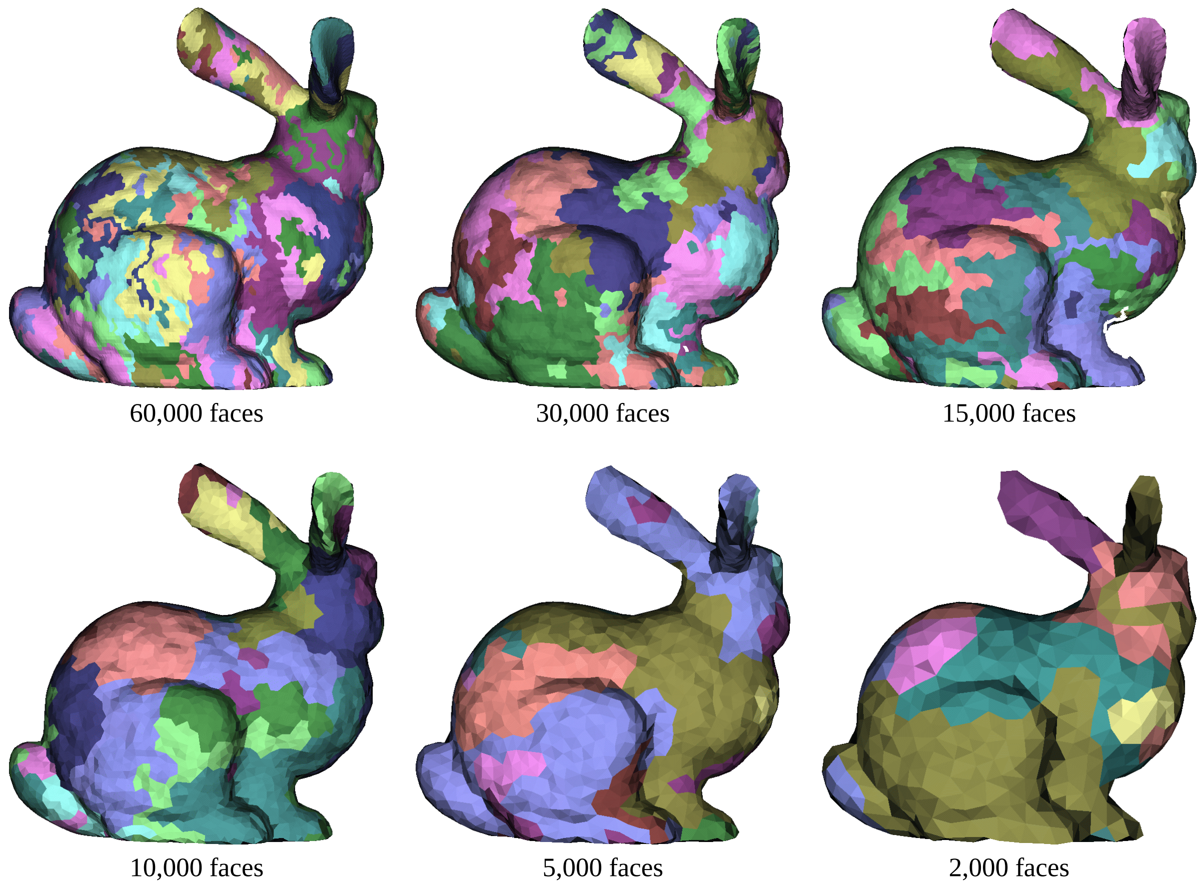


Figure 9: A progressive stripification of the 69,451–face Stanford bunny. Each coloured area is covered by one strip.

Table 1: A comparison of stripifications and execution times for the tunneling algorithm (restricted to tunnels of maximum length 75), the STRIPE 2.0 algorithm [7], and the SGI algorithm [1]. STRIPE failed for the two largest meshes. The bunny and dragon models are from the Stanford 3D Scanning Repository, while the random meshes are Delaunay triangulations of Poisson–distributed random point sets. Times are in seconds on a 750 MHz Pentium processor.

Model	Number of faces	Number of strips			Execution time		
		Tunneling	STRIPE	SGI	Tunneling	STRIPE	SGI
Random I	1,874	13	44	39	0.1	0.2	0.1
Random II	19,598	82	401	404	4.0	1.9	1.5
Bunny	69,451	158	917	705	17.4	9.5	10.3
Random III	198,734	593	—	3,719	522.5	—	84.4
Dragon	871,414	1,798	—	17,653	3.75 hours	—	0.45 hours

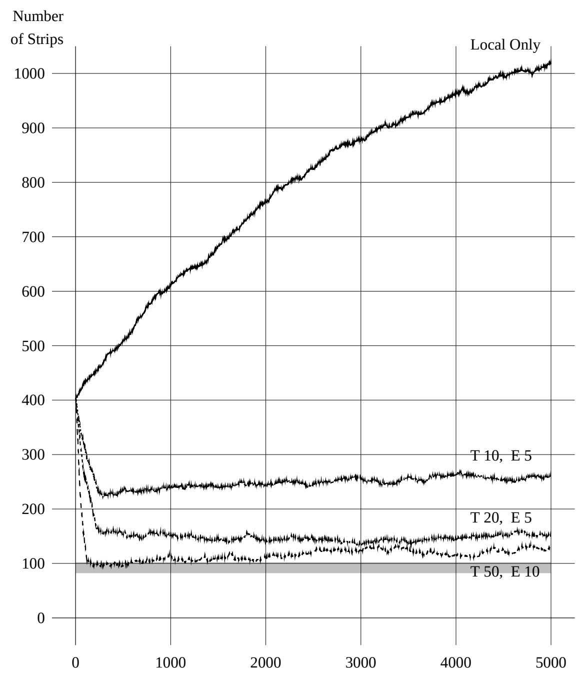


Figure 11: A non–tunneling "local–only" algorithm is compared to three variants of the tunneling algorithm over a sequence of 5000 collapse and split operations in a mesh of 20,000 faces, on average. The tunneling variants differ in the maximum allowed tunnel length (T 10, T 20, and T 50 for lengths of 10, 20, and 50) and in the number of strips from elsewhere in the mesh that are processed with each operation (E 5 and E 10 for 5 and 10 elsewhere). The grey region shows the lower bound of between 82 and 103 strips, achieved when the tunneling algorithm is run with no restrictions. As the tunnel length and the number of strips from elsewhere increase, the stripification approaches this lower bound.

4.3 Ordinary Progressive Meshes

A progressive mesh representation consists of a "base mesh" plus a sequence of vertex split operations. Application of the vertex splits to the base mesh recovers the original, high–resolution mesh.

In a similar manner one can encode a **progressive stripification**, which consists of a stripification of the base mesh plus a sequence of tunneling operations. The tunneling operations are performed in concert with the vertex splits to recover the original stripification of the high–resolution mesh. The tunneling operations can be chosen to yield high quality stripifications of the intermediate meshes. This is useful for progressive transmission, where transmission delays may require that the client render an intermediate version of the mesh before receiving the full sequence.

To create a progressive mesh representation, a sequence of edge collapses is computed which transforms the original mesh into the base mesh. This sequence is reversed and each edge collapse operation is replaced with its inverse vertex split operation, to yield the sequence of vertex splits.

The progessive stripification is built along with the progressive mesh. Each edge collapse of the progressive mesh is replaced by an **augmented edge collapse**:

1. Each edge collapse in the progressive mesh transforms a set of six nodes and five edges of the dual graph into a set of four nodes and two edges (recall Figure 2). Before the edge collapse is performed, the status of each of the five dual graph edges is recorded.

2. The edge collapse is performed.

3. After the edge collapse is performed, a tunneling operation is applied from each of the four remaining

Table 2: Statistics for progressive stripifications of three models, showing the average number of tunnel edges stored per PM edge collapse, and the average number of strips over all intermediate meshes.

Model	Faces	Collapses	Edges per collapse	Initial strips	Average strips
Random I	1874	996	2.91	18	12.3
Random II	19598	9996	3.51	104	76.8
Bunny	69451	34815	3.69	182	185.4

graph nodes, provided that the node is at the end of a triangle strip. The tunnel necessarily consists of a sequence of left and right turns through the graph (i.e. the tunnel arrives at a graph node on one edge and leaves the node on, alternately, the left or right of the remaining two edges), and can be concisely recorded.

To recover the original mesh and stripification from the base mesh and stripification, the sequence of augmented edge collapses is applied in reverse order. For each augmented edge collapse in the sequence:

1. The status of each edge on the tunnels is complemented.
2. The vertex split is performed.
3. The status of the five graph edges is restored.

The augmented edge collapse can be concisely represented: The status of the five graph edges requires five bits, and the tunneling operations can be recorded as a sequence of left and right turns through the graph, with a number of bits proportional to the path length. The maximum tunnel length can also be bounded in order to reduce the storage requirements.

Progressive stripifications were computed for three of the models, limiting the maximum tunnel length to 29 edges in order to accelerate the computation. Figure 9 shows some samples from the progressive stripification of the bunny. Table 2 shows that the extra storage required to represent the augmented edge collapses is relatively small: For example, only 3.69 tunnel edges must be stored per augmented edge collapse in the bunny model. Also, the table shows that the tunneling operations maintain good intermediate stripifications: The average number of strips (over all intermediate representations) is close to the initial number of strips, indicating that the augmented edge collapses do not cause much fragmentation.

5 Discussion

The tunneling operation is simple to implement. It can be used to build stripifications of static meshes, to maintain good stripifications of CLOD meshes, and to provide space–efficient progressive stripifications.

For static meshes, tunneling builds much better stripifications than the SGI and STRIPE algorithms, although it can take substantially longer in larger meshes, and should only be done off–line. For CLOD meshes, tunneling is the only method to date that maintains a good stripification during unrestricted topological changes: Unlike the Skip Strip approach, tunneling repairs triangle strips that become broken in intermediate mesh representations. For progressive meshes, tunneling permits the space–efficient transmission of good stripifications of all intermediate meshes.

6 Future Work

It is interesting to think of the tunneling operation as a rule in a graph transformation grammar. One could imagine further rules to improve stripifications. Such rules might, for example, exploit vertex cacheing or give precedence to sequential strips over generalized strips.

For generalized strips, a graph transformation rule might be developed to exploit strip loops, which occur occasionally within a stripification. Such loops are potentially very useful, since they can be broken anywhere with no increase in the number of strips. Breaking a loop provides two strip ends which can be used to join other triangle strips.

There are also situations in which multiple separate graph transformations, none of them a tunneling operation and none of them an improvement by itself, can in combination produce an improvement in the stripification.

It seems that there is a lot of potential in creating good stripifications by the application of graph transformation rules.

Acknowledgements

The author wishes to thank the reviewers for their many detailed and helpful comments, and wishes to thank the Stanford 3D Scanning Repository for the bunny and dragon models.

References

[1] K Akeley, P Haeberli, and D Burns. The tomesh.c program. Technical report, Silicon Graphics, 1990. Available on the SGI Developer's Toolbox CD.

[2] E. Arkin, M. Held, J. S. B. Mitchell, and S. Skiena. Hamiltonian triangulations for fast rendering. *The Visual Computer*, 12(9):429–444, 1996.

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

[4] M. Deering. Geometry compression. *Computer Graphics (SIGGRAPH)*, 29:13–20, 1995.

[5] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, and M. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *IEEE Visualization*, 1997.

[6] J. El-Sana, E. Azanli, and A. Varshney. Skip strips: Maintaining triangle strips for view-dependent rendering. In *IEEE Visualization*, pages 131–138, 1999.

[7] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *IEEE Visualization*, pages 319–326, 1996.

[8] S. Gumhold and W. Straßer. Real time compression of triangle mesh connectivity. *Computer Graphics (SIGGRAPH)*, 32:133–140, 1998.

[9] J. Hopcroft and R. Karp. An algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2:225–231, 1973.

[10] H. Hoppe. Progressive meshes. *Computer Graphics (SIGGRAPH)*, 30:99–108, 1996.

[11] H. Hoppe. View–dependent refinement of progressive meshes. *Computer Graphics (SIGGRAPH)*, 31:189–198, 1997.

[12] H. Hoppe. Optimization of mesh locality for transparent vertex caching. *Computer Graphics (SIGGRAPH)*, 33:269–276, 1999.

[13] M. Isenburg. Triangle Strip Compression. In *Graphics Interface*, pages 197–204, 2000.

[14] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner. Real-time continuous level of detail rendering of height fields. *Computer Graphics (SIGGRAPH)*, 30:109–118, 1996.

[15] M. McCool. Smash: A next-generation API for programmable graphics accelerators, API version 0.2. Technical Report CS-2000-14, University of Waterloo, August 2000.

[16] W. Schroeder, J. Zarge, and W. Lorensen. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH)*, 26:65–70, 1992.

[17] B. Speckmann and J. Snoeyink. Easy triangle strips for tin terrain models. In *Canadian Conference on Computational Geometry*, pages 239–244, 1997.

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

[19] C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface*, pages 26–34, 1998.

[20] L. Velho, L. H. de Figueiredo, and J. Gomes. Hierarchical generalized triangle strips. *The Visual Computer*, 15(1):21–35, 1999.

[21] X. Xiang, M. Held, and J. S. B. Mitchell. Fast and effective stripification of polygonal surface models. In *Interactive 3D Graphics*, pages 71–78, 1999.