Efficiently Generating Triangle Strips for Fast Rendering

Francine Evans Steven Skiena Amitabh Varshney

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

Abstract

Almost all scientific visualization involving surfaces is currently done via triangles. The speed at which such triangulated surfaces can be displayed is crucial to interactive visualization and is bounded by the rate at which triangulated data can be sent to the graphics subsystem for rendering. Partitioning polygonal models into triangle strips can significantly reduce rendering times over transmitting each triangle individually.

In this paper, we present new and efficient algorithms for constructing triangle strips from partially triangulated models, and experimental results showing these strips are about 15% better than those from previous codes. Further, we prove that it is NP-complete to find an optimal sequential triangulation. We also study the impact of larger buffer sizes and various queuing disciplines on the effectiveness of triangle strips.

1 Introduction

Interactive display rates are crucial to exploratory scientific visualization and virtual reality. The speed of high-performance rendering engines on triangular meshes in computer graphics can be bounded by the rate at which triangulation data is sent into the machine. Obviously, each triangle can be specified by three vertices, but to maximize the use of the available data bandwidth, it is desirable to order the triangles so that consecutive triangles share an edge. Using such an ordering, only the incremental change of one vertex per triangle need be specified, potentially reducing the rendering time by a factor of three by avoiding redundant clipping and transformation computations. Besides, such an approach also has obvious benefits in compression for storing and transmitting models.

Consider the triangulation in Figure 1. Without using triangle strips, we would have to specify the five triangles with three vertices each. By using triangle strips, as supported by the OpenGL graphics library [10, 11], we can describe the triangulation using the strip $(1, 2, 3, 4, 5, 6, 7, 8)$, and assuming the convention that the ith triangle is described by the ith, $(i+1)$st, and $(i+2)$nd vertices of the *sequential* strip. Such a sequential strip can reduce the cost to transmit n triangles from $3n$ to $n+2$ vertices.

In this paper, we consider the problem of constructing good triangle strips from polygonal models. Often such models are not fully triangulated, and contain quadrilaterals and

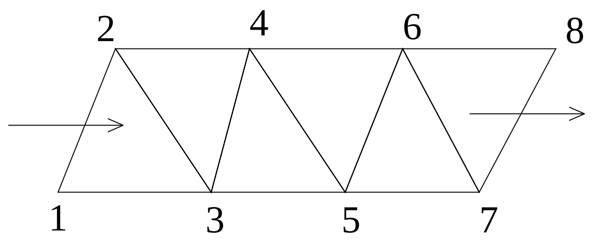


Figure 1: A Triangle Strip

other non-triangular faces, which must be triangulated prior to rendering. The choice of triangulation can significantly impact the cost of the resulting strips. For example, Figure 2 demonstrates that one triangle strip suffices to represent a cube, provided it is triangulated in a particular manner. In this paper we will prove that the problem of triangulating a polygonal model for optimal strips is NP-complete, and we will provide heuristics which exploit the freedom to triangulate these faces to produce strips that are $10 - 30\%$ better than those of previous codes. Our linear-time algorithm manages to achieve this by exploiting both the local and the global structure of the model. Our analysis of the global structure of a geometric model is done via a non-geometric technique we term *patchification*, which we believe is of general interest as an efficient tool for logically partitioning polygonal models.

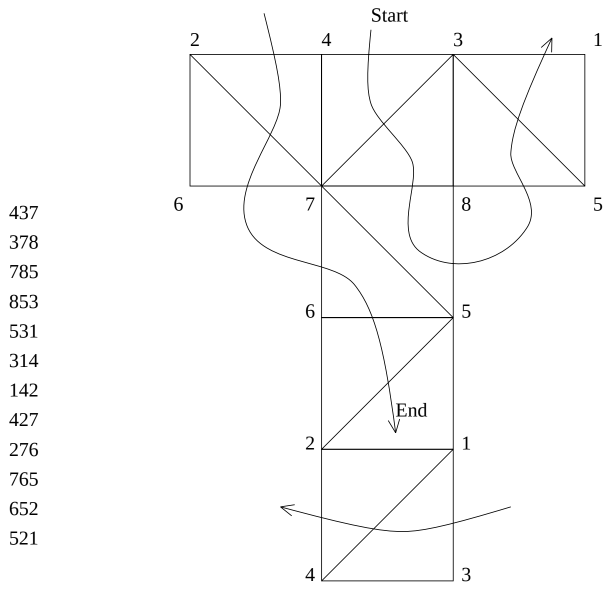


Figure 2: Triangulating a cube for one sequential strip.

To allow greater freedom in the creation of triangle strips, a "swap" command permits one to alter the FIFO queuing discipline in a triangle strip [13]. A swap command swaps the order of the two latest vertices in the buffer so that the instead of vertex i replacing the vertex $(i-2)$ in a buffer of size 2, vertex i replaces the vertex $(i-1)$. This allows for a single triangle strip representation of the collection of triangles shown in Figure 3, as $(1, 2, 3, SWAP, 4, 5, 6)$. This form of a triangle strip that includes swap commands is referred to as a *generalized triangle strip*.

The swap command gives greater freedom in the creation of triangle strips at the cost of one bit per vertex. Although the swap command is supported in the GL graphics library [13], keeping portability considerations in mind it was decided to not support it in OpenGL [7]. With OpenGL gaining rapid acceptance in the graphics software community, the one-bit-per-vertex cost model that was appropriate for a swap command in GL is now outdated. A more appropriate cost for such a swap command under the OpenGL model is a penalty of one vertex as explained next. One can simulate a swap command in OpenGL by re-transmitting the vertex that had to be swapped. This results in an empty triangle two of whose vertices are the same. This is illustrated in Figure 3, where we simulate $(1, 2, 3, SWAP, 4, 5, 6)$ by $(1, 2, 3, 2, 4, 5, 6)$. Note that, even though a swap costs one vertex in the OpenGL model, it is still cheaper than starting a new triangle strip that costs two vertices. In this paper, we evaluate all algorithms for both the GL and OpenGL cost models.

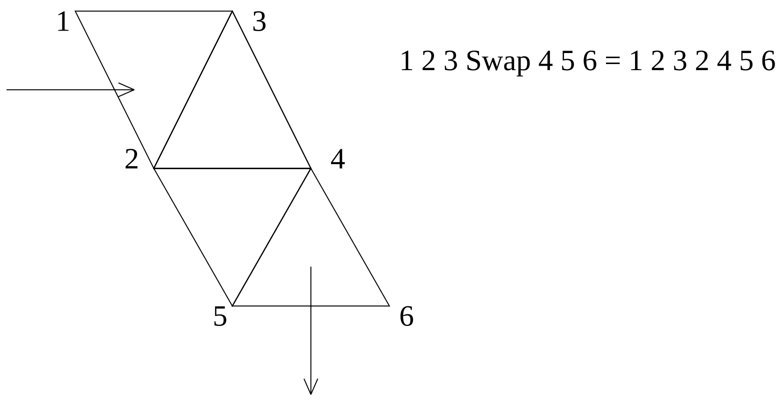


Figure 3: Replacing a swap requires an extra vertex.

Special-purpose rendering hardware is needed to fully exploit the advantages of triangle strips, by maintaining a buffer with the k previously transmitted vertices as determined by a certain queuing discipline. Although current rendering engines use a buffer of size of $k = 2$ and FIFO queuing discipline, there has been recent interest in studying the impact of larger buffer sizes, for both rendering [3] and geometric compression [6]. The decomposition of a triangular mesh into a triangle strip data structure that back-references the previous k vertices, $k \geq 2$ is referred to as a *generalized triangle mesh* [6]. Towards this end, we provide extensive analysis of the impact of buffer size and queuing discipline on triangle strip performance. We demonstrate that relatively small buffer sizes are sufficient to achieve most of the potential benefits of triangle strips, making for a desirable tradeoff between increasing hardware cost versus the speedup in rendering time.

In Section 2, we summarize previous work on triangular strips. In Section 3, we describe our local and global algorithms for constructing quality triangle strips from polygonal meshes. Experimental results are presented in Section 4. In Section 5, we prove that it is NP-Complete to find an optimal sequential triangulation. In Section 6, we study the impact of buffer size on triangle strip performance. Our conclusions are stated in Section 7.

2 Previous Work

The problem of constructing quality triangle strips has received attention from both the graphics and the computational geometry communities.

Akeley, Haeberli, and Burns have written a program that converts triangle meshes to triangle strips [1]. We discuss the approach in this program in greater details in Section 3. Deering has proposed the use of generalized triangle meshes for compressing connectivity information in geometric polygonal models [6]. He has proposed maintaining a stack of size $k = 16$ to store 16 previous vertices. A vertex for a new triangle is specified either through back-referencing one of the existing vertices on the stack, or by reading-in a new vertex and replacing an existing vertex on the stack. Although a novel idea, no algorithms have been proposed there to suggest how one can decompose polygonal models into generalized triangle meshes for a given buffer size k. An interesting alternative to compressing connectivity information is presented by Hoppe in [8] where vertex-split/edge-collapse information is encoded efficiently with respect to its neighbors. Although not as efficient as generalized triangle meshes for a single resolution model, this approach has the advantage of being able to encode multiresolution models compactly.

Within computational geometry, interest has focused on constructing and recognizing Hamiltonian and sequential triangulations. A triangulation is *Hamiltonian* if its dual graph contains a Hamiltonian cycle. Hamiltonian triangulations can be represented by using generalized triangle strips (triangle strips with swaps). Arkin, et.al. [2] proved that every point set has a Hamiltonian triangulation. Further, they showed that the problem of testing whether a triangulation is Hamiltonian is NP-complete. They gave an $O(n^2)$ algorithm for constructing a Hamiltonian triangulation of a polygon that has since been improved to $O(n \lg n)$ by Narasimhan [9].

A triangulation is *sequential* if its dual graph contains a Hamiltonian cycle whose turns alternate left-right. Sequential triangulations can be represented by using one triangle strip without any swaps. A Hamiltonian triangulation is sequential if three consecutive edges do not share a common vertex. Arkin, et.al. [2] proved that for any $n \geq 9$ there exists a set of n points in general position that do not admit a sequential triangulation. Although linear time suffices to test whether a triangulation is sequential, we will show in Section 5 that the problem of finding a sequential triangulation of a partially triangulated surface is NP-complete using a reduction from 3-satisfiability. Hence, heuristics such as those described in this paper are required to find good sequential strips.

A simple path in the dual of a triangulation identifies a sequence of triangles that form a "strip" or a (triangular) "ribbon" . Bhattacharya and Rosenfeld [4] have studied geometric and topological properties of ribbons. The Hamiltonian triangulation problem can be considered that of identifying if a set of points or a polygon has a triangulation that consists of a single strip (triangular ribbon).

Bose and Toussaint [5] have recently studied a set of problems involving *quadrangulation* of point sets, and have obtained several interesting results. A quadrangulation of a point set S is a decomposition of the convex hull into quadrilaterals, such that each point of S is a vertex of some quadrilateral. In particular, they have applied the notion of Hamiltonian triangulations to this problem, and they have obtained an alternate method of computing

Hamiltonian path triangulations.

By Euler's theorem on graphs, the number of triangles in a triangulation is at most twice the number of vertices, and on average we will have to send each vertex twice to the renderer using sequential triangle strips and a buffer of size 2. Bar-Yehuda and Gotsman [3] studied the extent to which we can increase the stack (buffer) size to reduce this duplication of vertices. This yields a time-versus-space tradeoff; for as we increase memory usage, rendering time will decrease. Bar-Yehuda and Gotsman have shown that a buffer of size $13.35\sqrt{n}$ is sufficient to render any mesh on n vertices in the optimal time n, and that a buffer size of $1.649\sqrt{n}$ is necessary for optimal rendering in the worst-case. They show the problem of minimizing the buffer-size for a given mesh is NP-hard, using a reduction from the problem of finding minimum separators of a planar graph.

3 Constructing Triangle Strips

In this section, we propose several heuristics for constructing triangle strips from polygonal models. There are at least three different objectives such heuristics might reasonably seek to achieve:

- *Maximize the length of each strip* – since each strip of length s represents $s-2$ triangles, maximizing strip length minimizes this overhead.
- *Minimizing swaps* – since each swap costs one additional vertex in the OpenGL cost model.
- *Minimizing the number of singleton strips* – since each triangle left isolated after removing a strip creates a singleton strip, we should seek to begin and end our strips on low-degree faces of the triangulation.

The best previous code for constructing triangle strips which we are aware of is [1], implementing what we will call the SGI algorithm. The SGI algorithm seeks to create strips that tend to minimize leaving isolated triangles. It is a greedy algorithm, which always chooses as the next triangle in a strip the triangle that is adjacent to the least number of neighbors (i.e. minimizes the number of *adjacencies*). When there is more than one triangle with the same, least number of neighbors, the algorithm looks one level ahead to its neighbors' neighbors, and chooses the direction of minimum degree, choosing arbitrarily if there is again a tie. After starting from an arbitrary lowest degree triangle, it extends its strips in both directions, so that each strip is as long as possible. There is no reluctance to generate swaps, and understandably so, since this algorithm was aimed at generating triangle strips for Iris GL.

A fast, linear-time implementation is obtained by using hash tables to store the adjacency information, linked to a priority queue maintaining strip length to choose which triangle starts a new strip.

Figure 4 illustrates how the algorithm breaks ties. Starting with a face of lowest adjacency (of degree 1 on the upper center of the figure), the algorithm always selects the lower degree

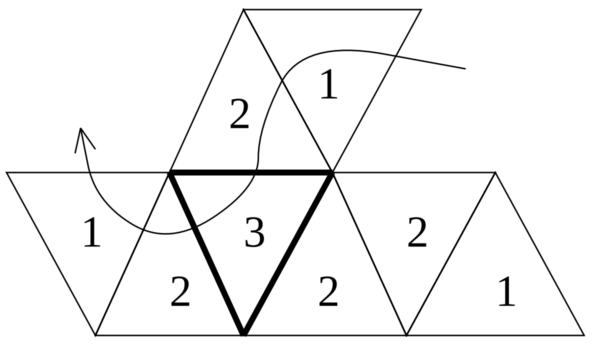


Figure 4: The SGI algorithm uses adjacency count to construct triangle strips.

face as the next triangle in the strip to peel off the marked strip. At the face of degree 3 it turns left because the adjacent face is of degree 1 as opposed to 2.

The SGI algorithm uses strictly local adjacency information in constructing the triangle strips. However, fully exploiting the freedom to triangulate quads seems to require a more global approach. We have experimented with several variants of local and global algorithms, as discussed in the following two sections.

3.1 Local Algorithms

Our class of local heuristics starts from the same basic idea as the SGI algorithm – to use least adjacencies as the basis for choosing the next face in a strip. However, we have tried to improve upon their algorithm by dynamic triangulation and alternate tie-breaking procedures.

We have considered three different approaches to triangulating faces:

- *Static triangulation* – In this approach, we triangulate all quads and larger faces in our model as a pre-processing step before we begin finding strips. We use alternate left-right turns, as shown in Figure 5(b) because such a triangulation is inherently sequential, as opposed to the simpler and more conventional fan triangulation. The SGI algorithm accepts only triangulated models as input and thus uses this static triangulation approach.

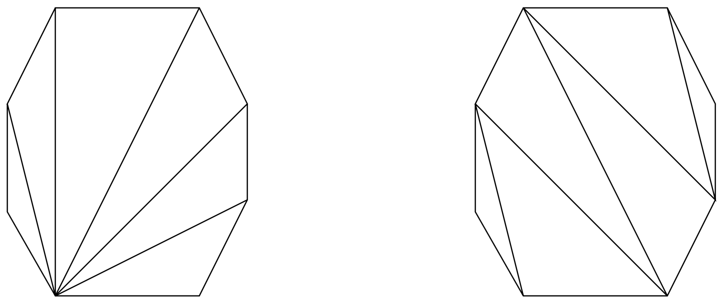


Figure 5: Fan versus sequential triangulation of a polygonal face.

- *Dynamic whole-face triangulation* – A second approach completely triangulates each face when we first enter it via some edge on a strip. After using one of the tie-breaking procedures described below to determine the exit edge e, we can triangulate the face as sequentially as possible while exiting at e. If the surface normals do not vary across a face, then whole face triangulation has the additional advantage of encoding fewer normal transitions.

- *Dynamic partial-face triangulation* – Partial-face triangulation provides the freedom to triangulate and walk only part of a face before exiting it. This approach can under certain conditions provably perform better than the whole-face triangulation, as is seen in the example where we represent a cube using a single sequential triangle strip. After identifying the exit edge e of the face with the minimum number of adjacencies, we sequentially triangulate the smallest portion possible of the face from the input edge to exit at e. This is illustrated in Figure 6.

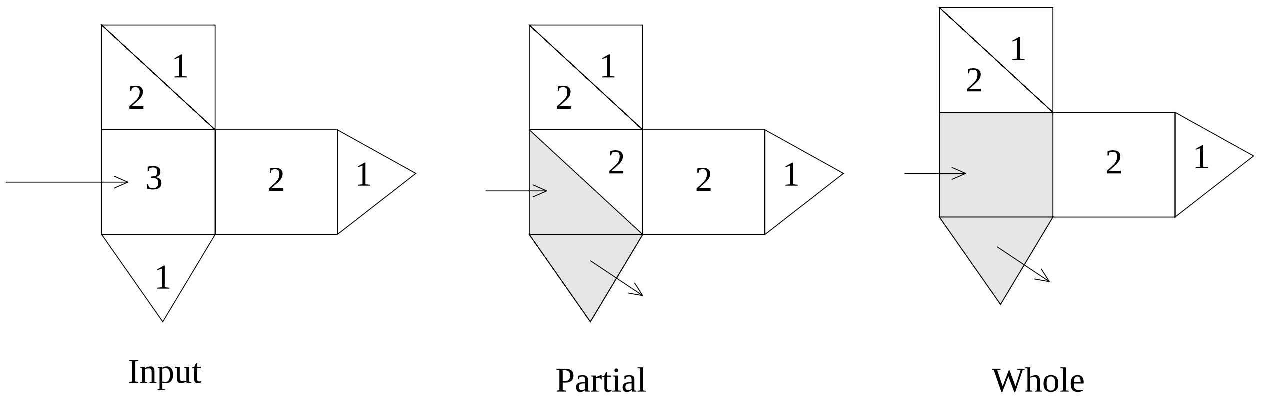


Figure 6: Examples of partial and whole-face triangulation.

We have considered several different approaches in breaking ties when there is more than one polygon that has the least number of adjacencies to the current face. Such ties often occur since the possible number of adjacencies ranges only over 1, 2, and 3. In particular, we tried:

- *Arbitrary* – meaning that we use the first face found among the low-adjacency faces.

- *Look-ahead* – this is the same approach that SGI algorithm takes, as described above.

- *Alternate* – this rule tries to alternate directions in choosing the next polygonal face. To motivate this option, note that sequential strips alternate directions.

- *Random* – chooses the next face randomly from those that were tied.

- *Sequential* – chooses the next face that will not produce a swap, and picks randomly if there is no such face.

To quickly identify the lowest adjacency face to start from, we maintain a priority queue ordered by the number of adjacent polygons to each face. The faces in the priority queue are linked to the adjacency list data structure representing the dual graph of the triangulation. This enables fast lookup to find and delete faces when forming the triangle strips.

3.2 Global Algorithms

Although the problem of finding the strip-minimal triangulation is NP-complete, we perform a global analysis of the structure of a polygonal model using a technique we call *patchification*, which we believe is of independent interest.

In typical polyhedral models, there are many quadrilateral faces, often arranged in large connected regions. We attempt to find large "patches", rectangular regions consisting only of quadrilaterals, as illustrated in Figure 7. Figure 8 shows the largest patches in a typical model. These patches can be triangulated sequentially along each row or column, although there is a cost of either 3 swaps per turn or 2 vertices to stop and restart each strip at the end of a row or column.

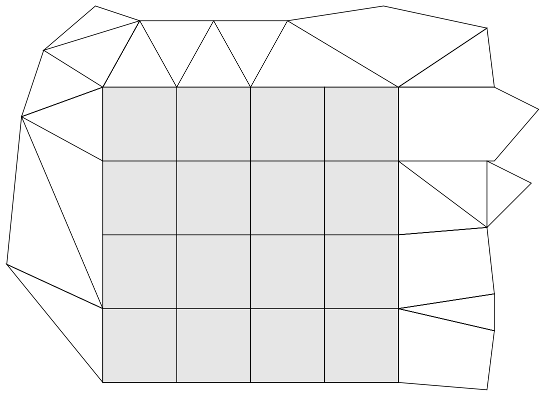


Figure 7: Patchification finds large rectangular patches of quadrilaterals.

Efficient patchification requires computing the number of polygons to the east, west, north, and south of each face, and making sure that when forming the patches, the polygons in the patch are all adjacent. Hence, we have to "walk" through the faces and calculate the number of adjacent polygons to them in each orientation. Each "walk" only visits each face exactly 2 times: once for the north-south direction and once for the east-west direction; once we visit a face in a walk, that face does not require visiting again. To avoid generating too many small patches, we keep a *patch cutoff size* which is the area of the smallest patch we would like to generate. Since we generate patches in decreasing order of size, we can conveniently stop the process once the areas of the patches being generated falls below this cutoff size. This approach takes us time $O(pn)$ where p is the number of patches found. In our studies p was much smaller than n and therefore this approach demonstrated a linear behavior.

We tried two different approaches for exploiting the coherence identified in large patches:

- *Row or column strips* – After selecting all patches whose size was greater than a specified cutoff size, we partitioned the patches into sequential strips along rows or columns

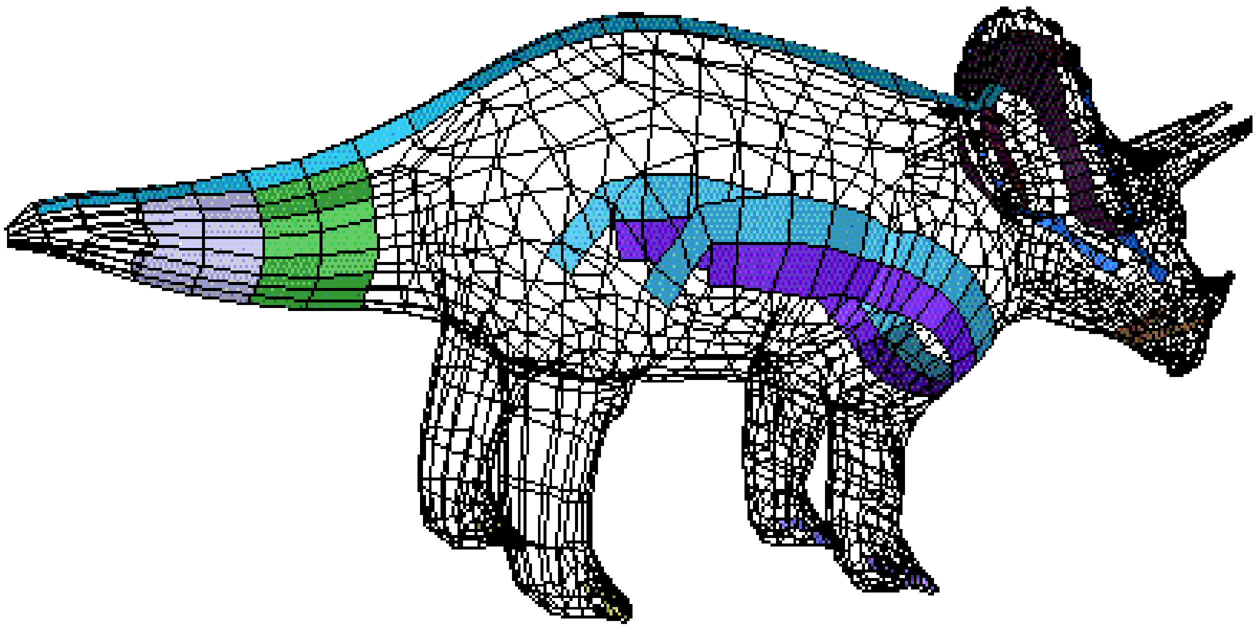


Figure 8: The six largest patches in a triceratops model.

(whichever direction yielded larger strips) and deleted them from the model. Next, a local algorithm (using whole-face triangulation) was used on the remaining model. By generating one strip along each row or column, we minimize the number of swaps needed.

- *Full-patch strips* – Each patch larger than the cutoff size was converted into one strip, at a cost of 3 swaps per turn. Further, every such strip was extended backwards from the starting quadrilateral and forwards from the ending quadrilateral of the patch to the extent possible. As before, the local algorithm was used on the model left after removing the patches and their forward and backward extensions.

4 Experimental Results

We have exhaustively tested our local and global algorithms on several datasets and compared them with the best known triangle strip code [1]. For our local approaches there were ten different options for each data file that we ran our experiments on: (a) whole-face triangulation and (b) partial-face triangulation, for each of the four tie breaking methods – (i) arbitrary, (ii) look-ahead, (iii) alternate, (iv) random, and (v) sequential. For our global approaches there were ten different options for each data file that we ran our experiments on: (a) row/column strips and (b) full-patch strips, for each of five different patch cutoff sizes of – 5, 10, 15, 20, and 25.

After comparing the results we had from the above-mentioned 20 different approaches on several datasets, we found that the best option was to use the the global row or column strips with a patch cutoff size of 5. We have implemented this option in our tool, *Stripe*. Table 1 compares the results for Stripe against the SGI algorithm. The cost columns show the total number of vertices required to represent the dataset in a generalized triangle strip representation under the OpenGL cost model (where each swap costs one vertex).

Data File	Vertices	Triangles	SGI Cost	Stripe Cost	% Savings
plane	1508	2992	4005	3509	12%
skyscraper	2022	3692	5621	4616	18%
triceratops	2832	5660	8267	6911	16%
power lines	4091	8966	12147	10621	13%
porsche	5247	10425	14227	12367	11%
honda	7106	13594	16599	15075	9%
bell ranger	7105	14168	19941	16456	17%
dodge	8477	16646	20561	18515	10%
general	11361	22262	31652	27702	12%

Table 1: Comparison of triangle strip algorithms on representative models.

Figures 9, 10, and 11 show the performance comparisons between our best local and best global algorithms against the SGI algorithm for (a) GL and (b) OpenGL cost models. The models sorted by number of triangles are along the x-axis and the cost of generalized triangle strip representation is along the y-axis in these figures.

Observations include:

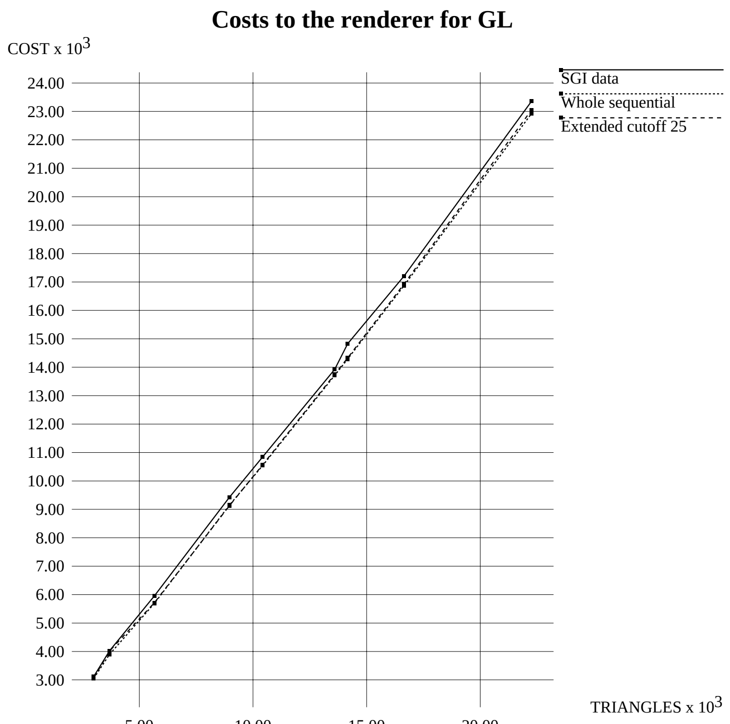
- Little if any savings seems possible by sophisticated algorithms under the GL model. However, under the more realistic model the combined local/global algorithm can save up to 20% over the SGI algorithm.

- Our results are close to the theoretical lower bound of the number of triangles + 2, so there is limited potential for better algorithms.

- Although the number of strips and number of swaps required is sensitive to the composition of the model, the total cost grows linear in the size of the model.

Our times for execution of these algorithms behaved linearly with respect the input size. The timings for our local algorithms were about a factor of 2 slower than those generated by SGI. Thus, for example, dynamic partial-face method with sequential triangulation took around 8 seconds on the $22K$ triangle model `general` whereas the SGI code took around 4 seconds.

When rendering the models with the triangle strips that were produced by each algorithm, the savings in transmission time to the renderer did prove to be a significant savings in rendering time. The triangle strips produced by our code were on average 30% faster to draw than those produced by the SGI algorithm, and were about 60% faster to draw than without using triangle strips at all. These savings increased as the size of the model increased, as shown in Table 2.



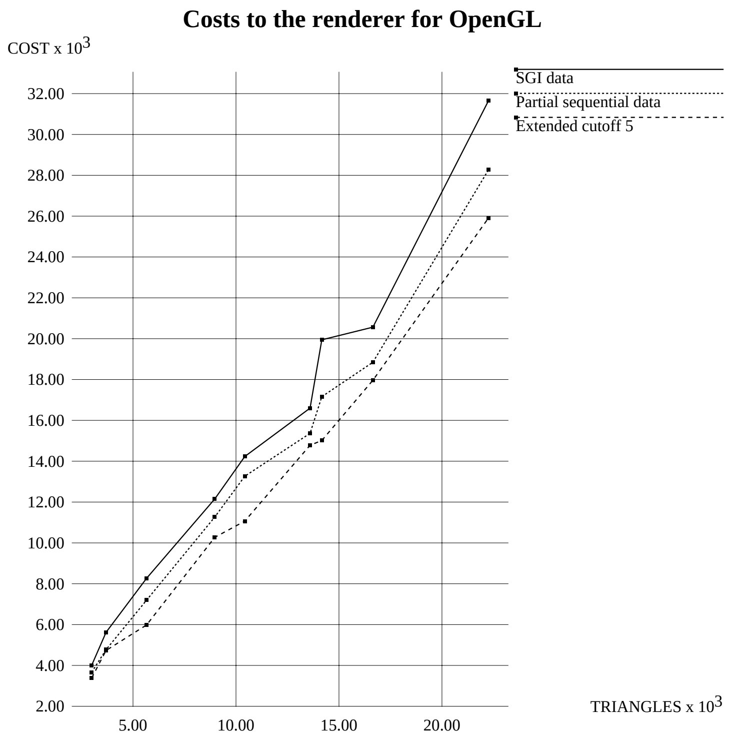
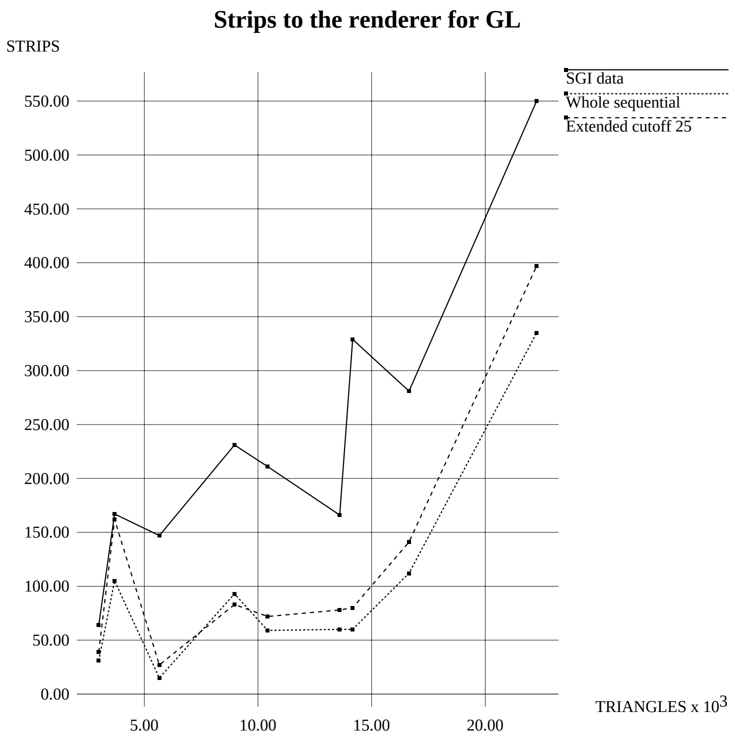


Figure 9: Overall cost comparisons for GL and OpenGL cost models.



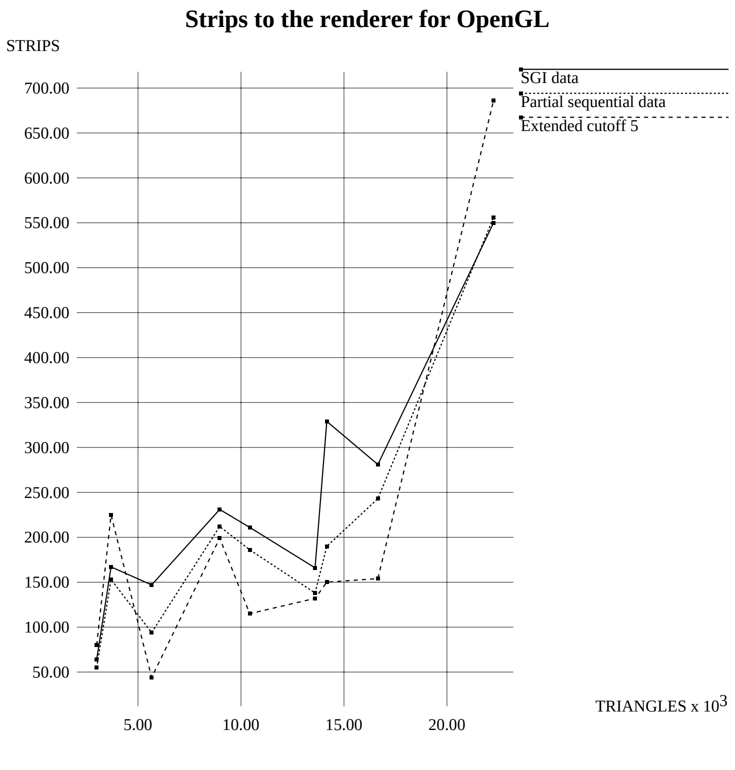
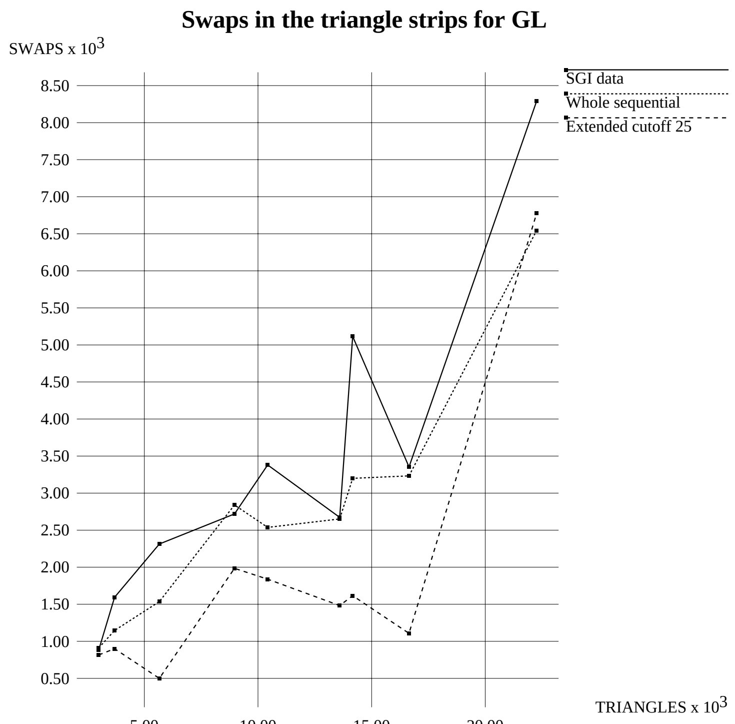


Figure 10: Number of strips produced for GL and OpenGL cost models.



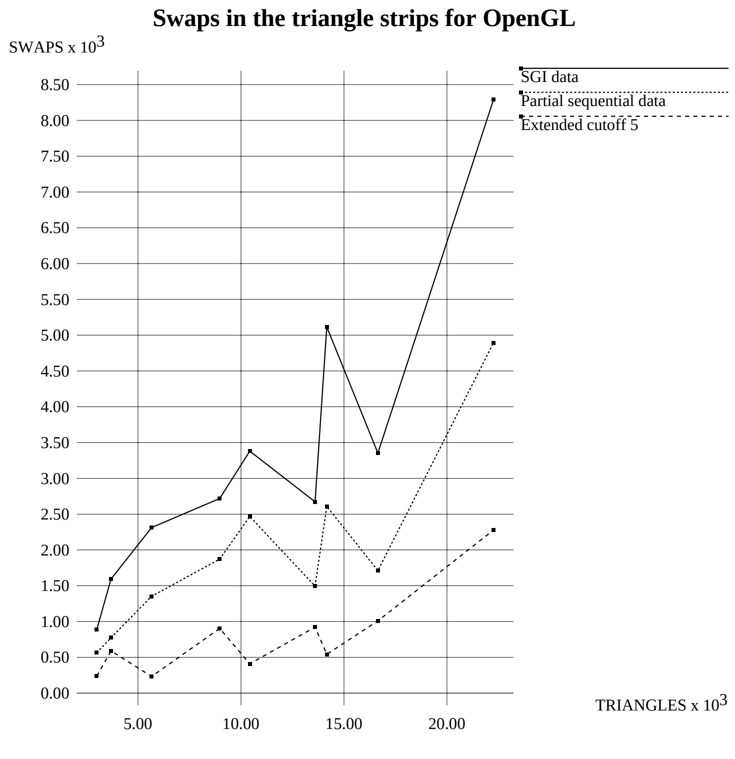


Figure 11: Total swaps produced for GL and OpenGL cost models.

Machine	File	Triangles	Triangulated	SGI	Ours
SGI Indigo2	Triceratops	5660	0.5	0.3	0.27
	Bell Ranger	14168	1.62	0.8	0.59
	General	22262	2.52	1.2	0.88
PC- 150MHz	Triceratops	5660	0.86	0.6	0.56
	Bell Ranger	14168	1.93	1.2	1.0
	General	22262	3.13	2.4	1.89

Table 2: Comparison of rendering times in seconds.

For local algorithms under the GL cost model whole-face triangulations worked better than those with partial-face triangulations; under the OpenGL cost model the reverse was true. Partial-face triangulations produce less swaps than whole-face triangulations because the former have a greater choice in selecting the next face in a strip, and are therefore more likely to be able to select faces that do not require a swap. For global algorithms, full-patch strips with cutoff size of 25 have the best performance under the GL cost model whereas full-patch strips with a cutoff size of 5 have the best performance under the OpenGL cost model. This is because a cutoff size of 5 generates more patches than a cutoff size of 25 and more patches means lesser number of swaps.

Figure 12 provides visual comparison of the results obtained by our tool Stripe and those obtained by the earlier algorithm being used by SGI.

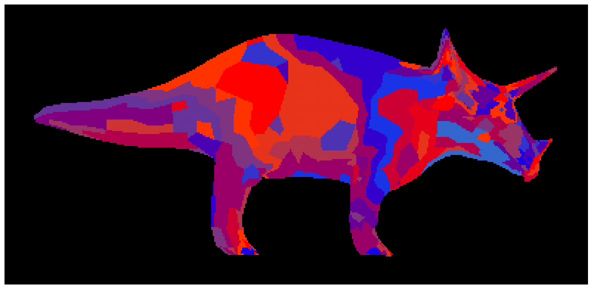
5 Sequential Triangulation Completion is NP-Complete

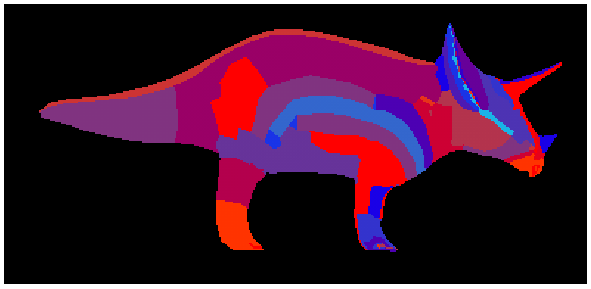
In this section, we consider the complexity of two problems associated with completing a partial triangulation so that it has a certain path property. In particular, we are interested in:

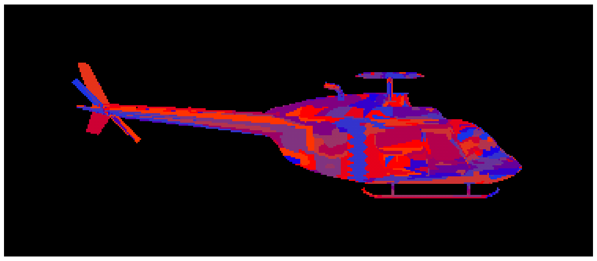
Hamiltonian Triangulation Completion
Input: A partially triangulated n-vertex polygon P (with holes).
Output: Is there a way to complete the triangulation of P such that the triangulation is Hamiltonian?

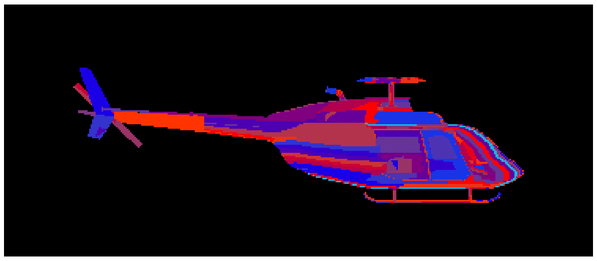
Sequential Triangulation Completion
Input: A partially triangulated n-vertex polygon P (with holes).
Output: Is there a way to complete the triangulation of P such that the triangulation is sequential?

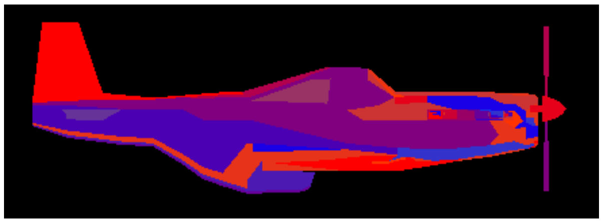
The hardness of Hamiltonian Triangulation Completion follows immediately from the result of [2] that it is NP-complete to decide whether a full triangulation is Hamiltonian. However, establishing the complexity of Sequential Triangulation Completion proves considerably more subtle, because sequential triangulations are highly constrained. Indeed, the

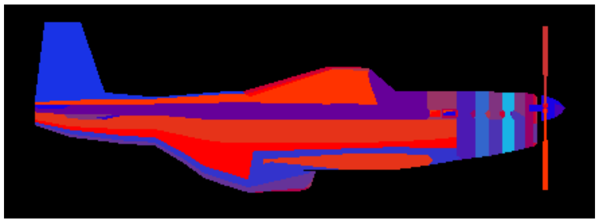

SGI

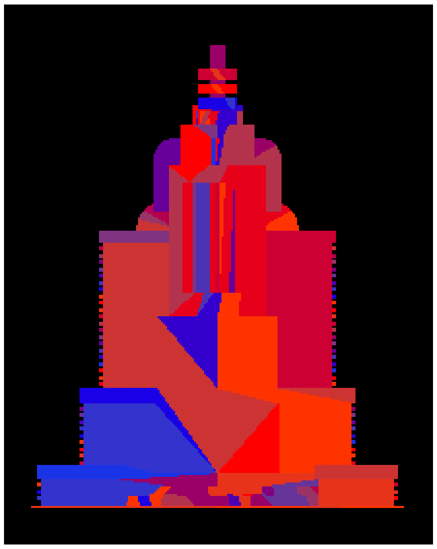

Stripe


SGI


Stripe


SGI


Stripe


SGI

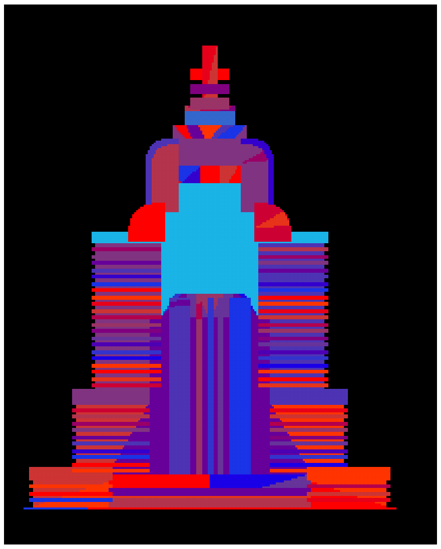

Stripe

Figure 12: Visual Comparison of Triangle Strips Generated by SGI and Stripe

corresponding problem of testing whether a full triangulation is sequential can be solved in $O(n)$ time [2].

We prove that Sequential Triangulation Completion is NP-complete with a reduction from Hamiltonian path in directed graphs with vertex degree three, shown hard by Plesnik [12]. A sketch of our proof is as follows. First, we show how to construct polygonal gadgets to play the role of low-degree vertices in a directed graph. Second, we will show that these polygonal gadgets can be used to construct polygonal implementations of the graph gadgets used in Plesnik's hardness proof. Finally, we show how to modify these polygonal gadgets so that any Hamiltonian paths in the graph will correspond to sequential paths in the triangulation.

For the first phase of our proof, we construct polygonal gadgets corresponding to 1-input 2-output and 2-input 1-output vertices of a directed graph. These gadgets will be described in Lemmas 1 and 2, respectively. We say an edge e is *oriented* (x, y) in a sequential path p if vertex x appears immediately before vertex y in p.

Lemma 1 *For every $n \geq 4$, there exists a partially triangulated convex n-gon with consecutive vertices labeled $(1, \ldots, n)$ and an interior point v which can be sequentially triangulated in the following two different ways. For even n, the triangulations enter oriented edge $e = (1, 2)$ but exit oriented edges $a = (n/2 + 3, n/2 + 2)$ and $b = (n/2 + 1, n/2)$. For odd n, the triangulations enter oriented edge $e = (1, 2)$ but exit oriented edges $a = (\lceil n/2 \rceil + 1, \lceil n/2 \rceil + 2)$ and $b = (\lceil n/2 \rceil - 1, \lceil n/2 \rceil)$. (All arithmetic is modulo n.)*

Proof: The smallest case for even n is $n = 4$. Consider the polygon P in Figure 13, with hole vertex v near edges $(1, 2)$ and $(2, 3)$. We can sequentially triangulate P with 1 input and 2 outputs, so that a=(1,4) and b=(3,2), as shown in Figure 13. For larger even n, we can simply prepend pairs of triangles to the input face of this gadget.

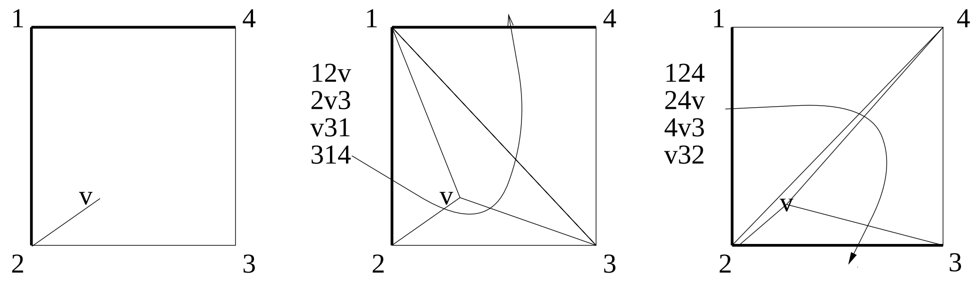


Figure 13: The 1-input, 2-output gadget for $n = 4$.

The smallest case for odd n is $n = 5$. Consider the polygon P and triangulations in Figure 14. Due to the odd number of triangles, the output edge orientations will be reversed from the even case. ∎

Lemma 2 *For every $n \geq 4$, there exists a partially triangulated convex n-gon with consecutive vertices labeled $(1, \ldots, n)$ with hole which can be sequentially triangulated in the following two different ways. The triangulations enter oriented edges $a = (1, 2)$ and $b = (3, 4)$. For even n, the triangulation exits edge $e = (3 + n/2, 2 + n/2)$. For odd n, the triangulation exits edge $e = ((3 + n)/2, (3 + n)/2 + 1)$. (All arithmetic is modulo n.)*

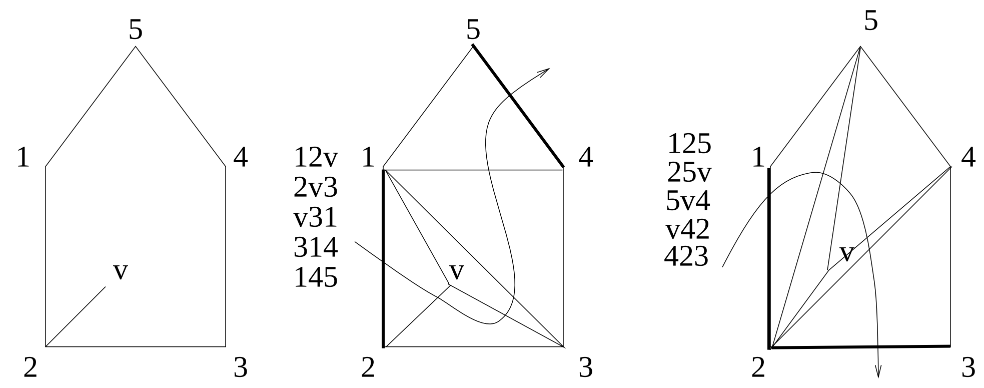


Figure 14: The 1-input, 2-output gadget for $n = 5$.

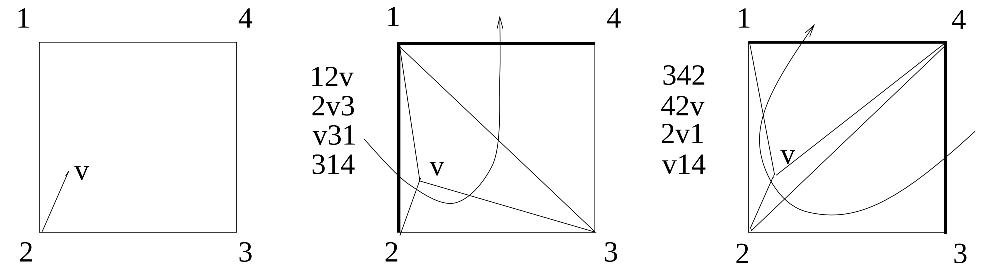


Figure 15: The 2-input, 1-output gadget for $n = 4$.

Proof: This proof is analogous to that of Lemma 1. The smallest even case is when $n = 4$, and shown in Figure 15. The construction generalizes to larger even n by appending pairs of triangles to the output edge. To realize the construction for odd n, append an odd number of triangles to the output edge, as in Figure 16. ▮

For the second phase of our reduction, we mirror the three types of gadgets used by Plesnik in showing that the directed Hamiltonian path hard for vertex degree-3 digraphs. Figures 17, 18, and 19 illustrate these components, and demonstrate that they can be replicated with abutting and non-crossing convex polygons in the plane, each of which contains the appropriate vertex replacement gadget from Lemmas 1 and 2.

Thus we replicate Plesnik's reduction yielding a partially triangulated polygon with holes

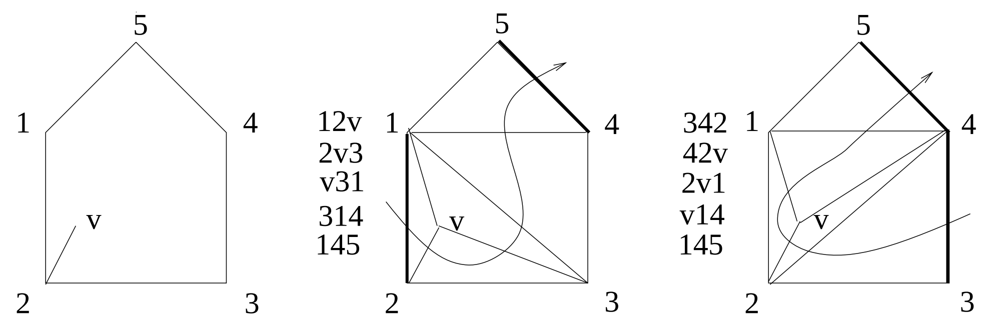


Figure 16: The 2-input, 1-output gadget for $n = 5$.

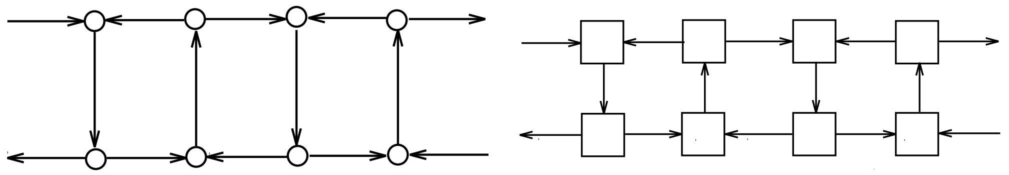


Figure 17: Plesnik gadget 1, as a graph and in the plane.

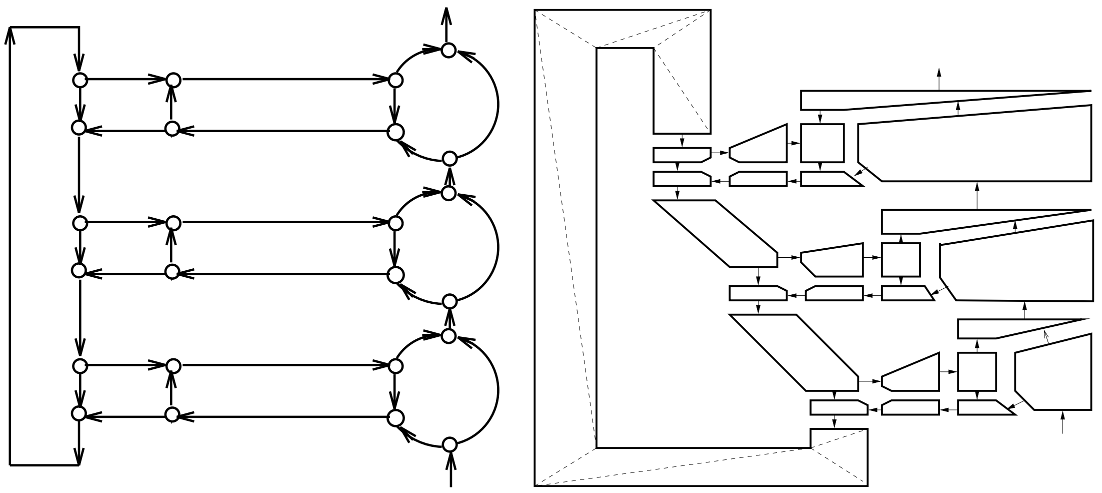


Figure 18: Plesnik gadget 2, as a graph and in the plane.

which can have a Hamiltonian triangulation iff the original graph was Hamiltonian.

Corollary 3 *Hamiltonian triangulation completion is NP-complete.*

Unfortunately, this construction need not yield a sequential triangulation, due to orientation inconsistencies at the junctions of these gadgets. Resolving these problems will require modifications to the above gadgets and the following observation:

Lemma 4 *The outgoing edges of any 1-input vertex in Plesnik's construction are adjacent to 2-input vertices. The outgoing edge of any 2-input vertex is adjacent to a 1-input vertex.*

Hence any path in the resulting graph must alternate between 1 and 2-input vertices.

To correct sequential inconsistencies at the junctions of the gadgets, we first assign orientations to the unique starting and terminating edges of the Hamiltonian path construction. We then compare the orientations on both sides of each gadget-junction in the construction. The junctions between the original 1-input and 2-input gadgets described above may be inconsistent. Indeed there can be six possible types of inconsistencies between gadgets. Four of them are easily resolved by adding single vertices to the gadgets:

- *2-input, 1-output gadget, with the output edge reversed* – Raising the output edge into a triangle with one additional vertex corrects the orientation, as shown in Figure 20.
- *2-input, 1-output gadget, with both input edges reversed* – The gadget in Figure 21 resolves the difficulty.

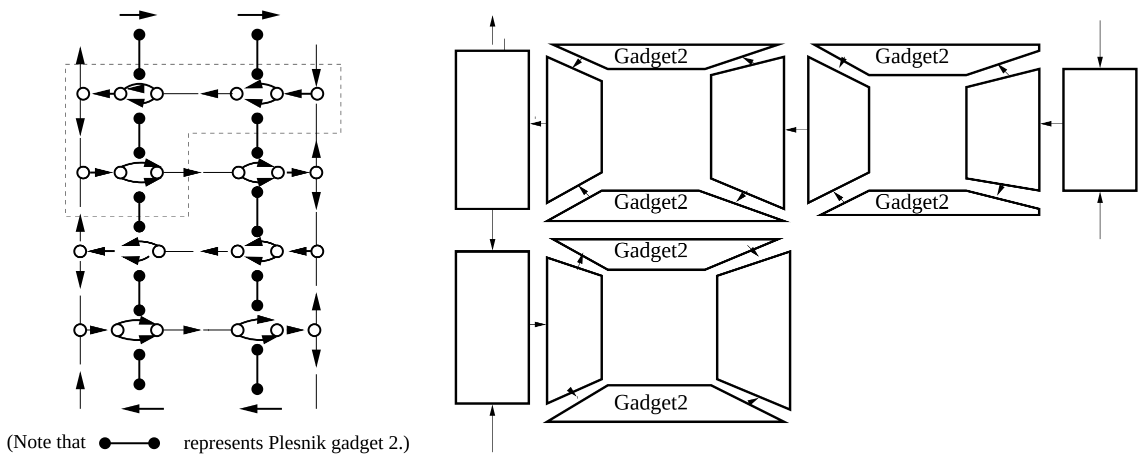


Figure 19: Plesnik gadget 3, as a graph and in the plane. Note that the dotted region in the gragh is the region that is represented in the plane.

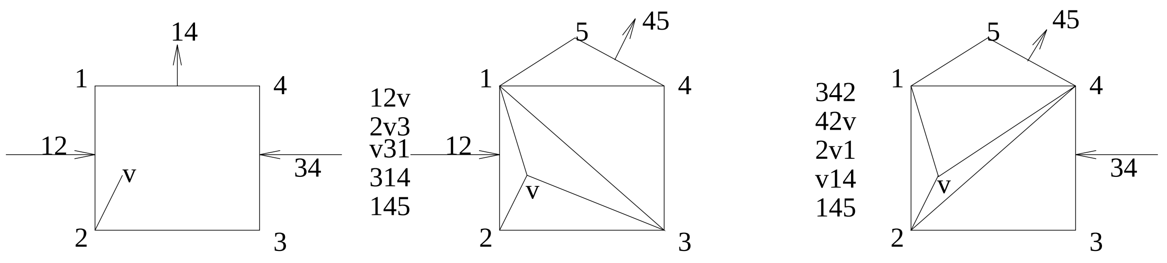


Figure 20: Reversing the output orientation of a 2-Input, 1-Output gadget.

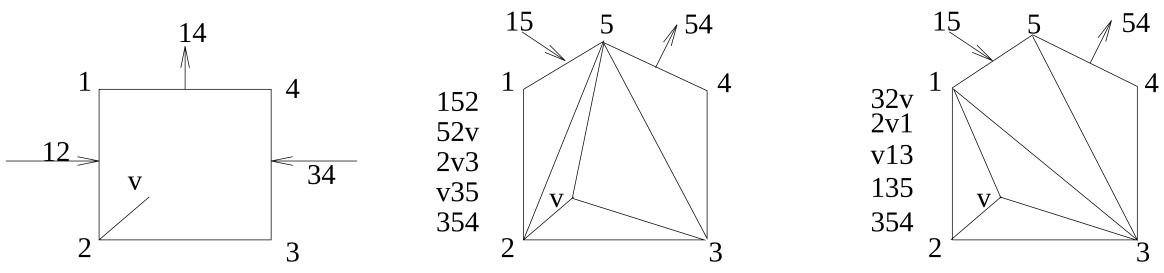


Figure 21: Reversing the orientation of both inputs of a 2-Input, 1-Output gadget.

- *1-input,2-output gadget, with the input edge reversed* – Raising the input edge into a triangle with one additional vertex corrects the orientation, as shown in Figure 22.

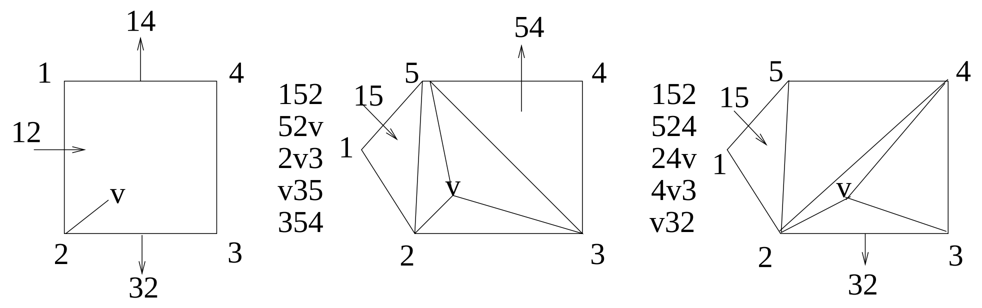


Figure 22: Reversing the input orientation of a 1-Input, 2-Output gadget.

- *1-input,2-output gadget, with both output edges reversed* – The gadget in Figure 23 resolves the difficulty.

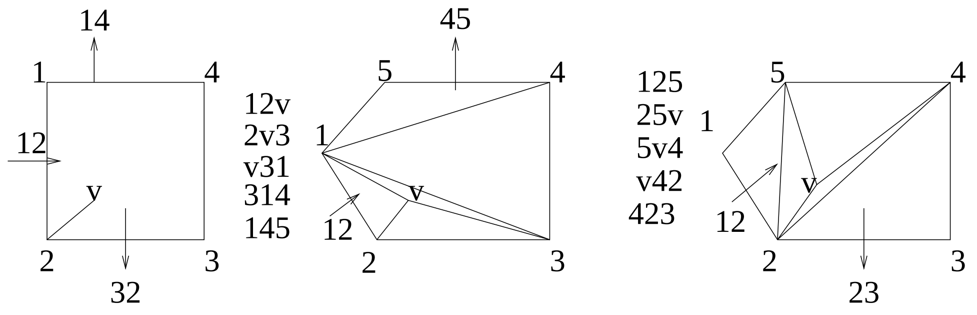


Figure 23: Reversing the orientation of both outputs of a 1-Input, 2-Output gadget.

After performing these corrections, the only possible remaining inconsistent orientations are one wrong input to a 2-input gadget or one wrong output from a 2-output gadget. However, by Lemma 4, gadgets must alternate input sizes, and so these inconsistent gadgets must occur in pairs, with a 1-input gadget feeding into a 2-input gadget.

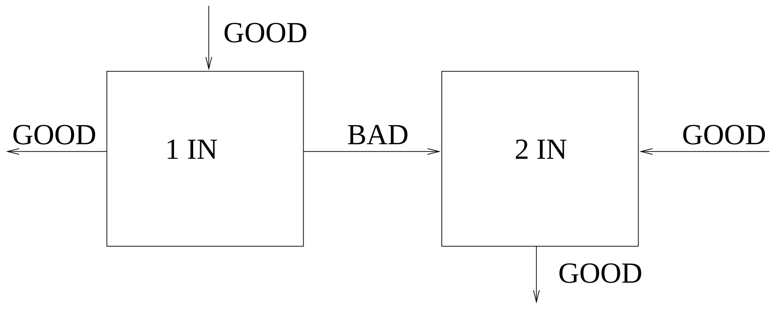


Figure 24: The remaining case of inconsistent orientations.

We can replace this pair as in Figure 24 by one gadget that resolves the inconsistency and still maintains the orientations of the external inputs and outputs. To replace this pair, we use the gadget in Figure 25. This combined gadget is shown in Figure 26. In the interests of brevity, we omit a detailed case analysis, but these are the only way to cover the gadget with possible entries and exits from the connection edges. As no other faces of the gadget will be adjacent to other gadgets, any Hamiltonian path must cover the gadget in one of these two ways.

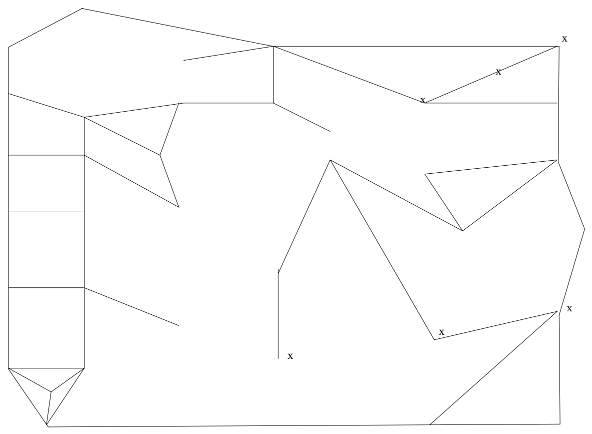


Figure 25: The combined gadget with all the forced edges drawn in. The points marked with "x" are points that are collinear and form degenerate triangles in the two possible triangulations.

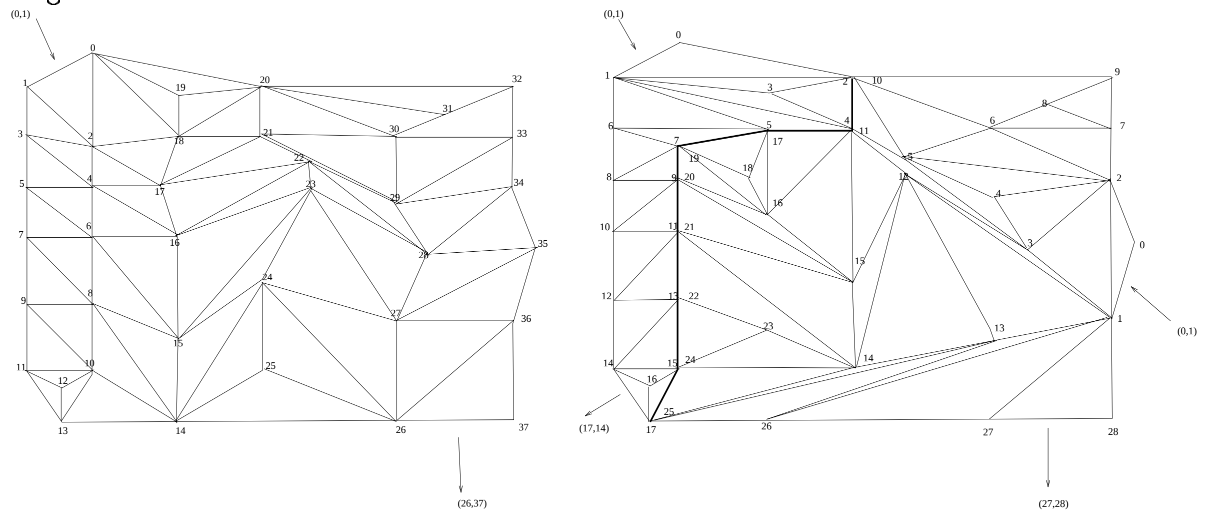


Figure 26: The two possible triangulations of the combined gadget.

In conclusion:

Theorem 5 *Sequential triangulation completion is NP-complete for polygons with holes, even when no untriangulated face has more than 24 vertices.*

6 Impact of Buffer Size

The benefits realized by using triangle strips could be further enhanced by special-purpose hardware that has additional buffer space (beyond the usual storage for two vertices) and alternate queuing disciplines. In this section, we study the impact of such resources on performance, to provide guidance for future hardware design.

Increasing the buffer size from a capacity of two vertices naturally decreases the cost of transmission, since we can now specify which of the previous k vertices in the buffer defines the next triangle. The cost of specification becomes $\lceil \lg k \rceil$ bits, instead of number of bits representing one vertex, thus enabling us to potentially represent polygonal models at a cost of less than one vertex per triangle. In our paper, we will ignore the costs of these index bits, since we only seek to determine an upper bound potential improvement in rendering time to assess whether it might be worth the increase in hardware costs.

We considered two different queuing disciplines for maintaining the buffer:

- *First-in, first-out (FIFO)* – This implies that there is no rearrangement of the vertices in the buffer, excluding swaps. FIFO is easiest to implement in hardware, and would thus be preferable if performance is comparable.
- *Least recently used (LRU)* – LRU dynamically rearranges the vertices in the buffer, by placing a vertex that was used most recently into the spot in the buffer that holds the most recently admitted vertex. The least recently used vertex is eliminated when a new vertex is added to the queue. LRU provides the benefit that popular vertices are held in the buffer in the hope that they will likely be used in the near future.

The results of running our tests on several datasets are presented in Figure 27. These figures show the cost of the LRU and FIFO queuing disciplines versus the dataset sizes. As can be seen the advantages to be gained from larger buffer sizes diminish rapidly beyond a buffer size of about 8. For this range of buffer sizes, LRU performs better than the FIFO scheme by a factor about 10%.

7 Conclusions

We have explored a total of twenty different local and global algorithms on over two hundred data models in our quest for an effective triangle strip generation algorithm that can perform well under the prevalent OpenGL cost model. Our conclusion is that the best approach for the OpenGL cost model is global row or column strips with a patch cutoff size of 5. Our source code is freely available from http://www.cs.sunysb.edu/~evans/stripe.html.

As can be seen from the results of Table 1, we are able to outperform the SGI algorithm significantly. We typically produce a significantly lower number of strips than they do (usually 60%-80% less using the local whole-triangulation algorithm), resulting in an average cost savings of about 18% less than SGI algorithm under the OpenGL model. Further, our cost averages just 10% more than the theoretical minimum of using one sequential strip with no swaps, when using the global row or column strips algorithm with a patch cutoff size of 5, as shown in Figure 9.

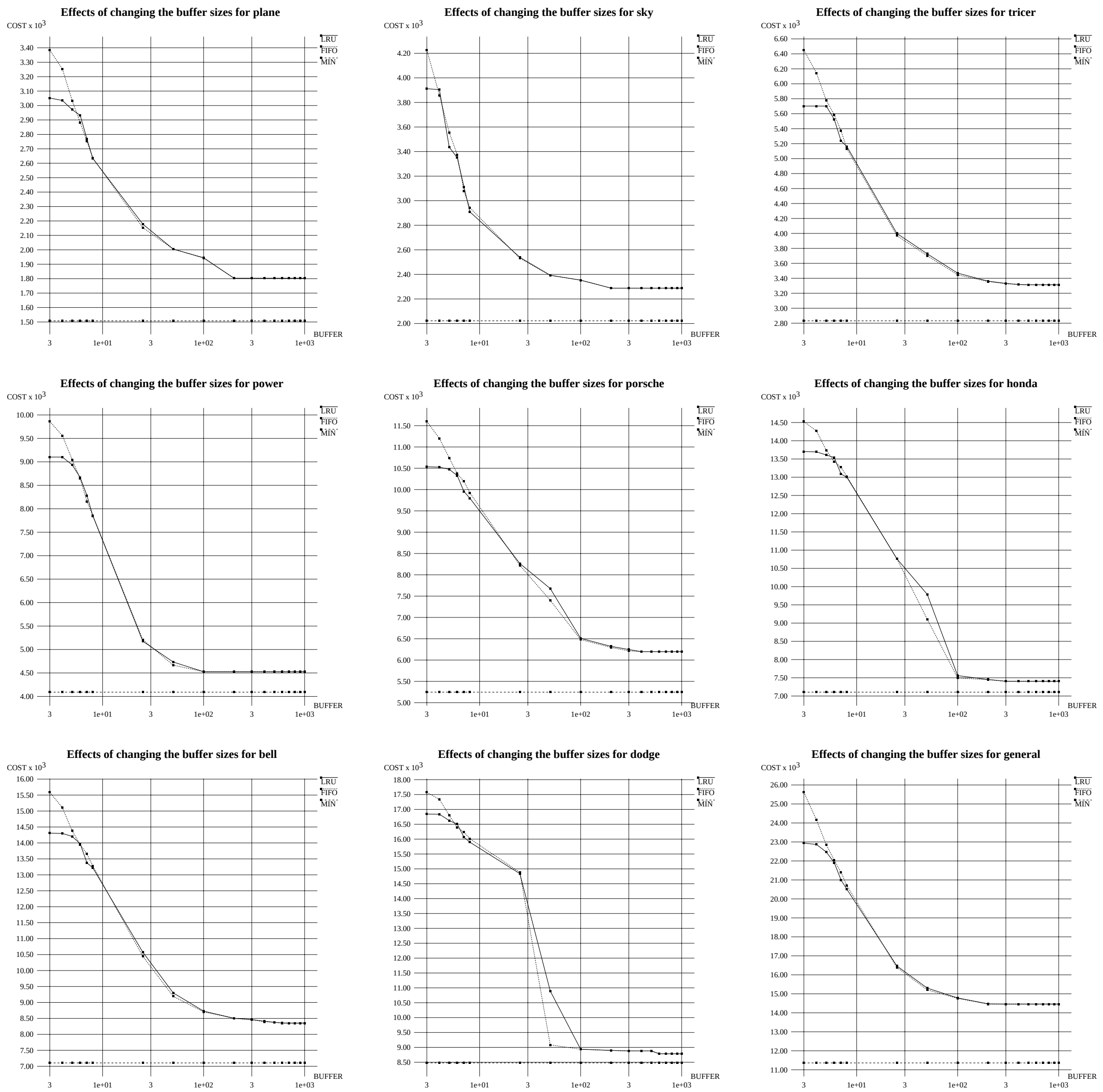


Figure 27: Cost versus buffer size for nine models.

We have found that using global algorithms for detecting large strips of quads proves very effective for reducing swaps. This has proved to be quite useful for generating efficient triangle strips for the OpenGL cost model where every swap costs one vertex.

All our algorithms run in linear time. Although the SGI algorithm does have a slightly better running time, we do not believe this to be a serious drawback of our approach since the triangle-strip generation phase is typically done off-line before interactive visualization. Also, our algorithm can take as input a polygonal model, while the SGI algorithm cannot handle polygonal data. Therefore to use their algorithm, the user needs to first pre-triangulate the data model, which is an extra step not added into the SGI running time.

The results of our experiments with larger buffer sizes offer only limited room for optimism. As we increase the buffer-size the savings do increase, however the improvements diminish very quickly. LRU seems to work much better than FIFO in the smaller buffers, although this must be contrasted with the time and hardware needed to maintain a LRU buffer. The theoretical minimum of using larger buffers is the number of vertices in the model, since each vertex would only have to be transmitted exactly one time, and then could remain in the buffer forever to be used again, provided the buffer is large enough. However, in our implementation we had been assuming that the buffer gets flushed between renderings of different generalized triangle meshes, i.e. a generalized triangle mesh cannot take advantage of the buffer references left behind by a previous mesh. Even if we do not make this assumption, achieving close to the minimum requires a prohibitively large buffer, which is not feasible for hardware implementation. Further, as the result of Bar-Yehuda and Gotsman [3] shows, to achieve this minimum for a mesh of size n a buffer of size $1.649\sqrt{n}$ is necessary, thus making the size of the buffer depend on the size of the input mesh. All of these factors combined with our results seem to make a choice of a small buffer size, say around 8, attractive.

Acknowledgements

We would like to acknowledge several valuable discussions we have had on triangle strips with Joe Mitchell, Martin Held, Estie Arkin, Jarek Rossignac, Josh Mittleman, and Jim Helman. The datasets that we have used have been provided by Viewpoint DataLabs. Francine Evans is supported in part by a NSF Graduate Fellowship and a Northrop Grumman Fellowship. Steven Skiena is supported by ONR award 400x116yip01. Amitabh Varshney is supported in part by NSF Career Award CCR-9502239.

References

[1] K. Akeley, P. Haeberli, and D. Burns. tomesh.c : C Program on SGI Developer's Toolbox CD, 1990.

[2] E. Arkin, M. Held, J. Mitchell, and S. Skiena. Hamiltonian triangulations for fast rendering. In *Second Annual European Symposium on Algorithms*, volume 855, pages 36–47. Springer-Verlag Lecture Notes in Computer Science, 1994.

[3] R. Bar-Yehuda and C. Gotsman. Time/space tradeoffs for polygon mesh rendering. *ACM Transactions on Graphics*, 15, no. 2:141–152, 1996.

[4] P. Bhattacharya and A. Rosenfeld. Polygonal ribbons in two and three dimensions. Technical report, Department of Computer Science, University of Maryland, 1994.

[5] J. Bose and G. Toussaint. No quadrangulation is extremely odd. Technical Report 95-03, Department of Computer Science, University of British Columbia, 1995.

[6] M. Deering. Geometry compression. *Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH*, pages 13–20, 1995.

[7] J. Helman. Personal Communication.

[8] Hugues Hoppe. Progressive meshes. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 99–108. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[9] G. Narasimhan. On Hamiltonian triangulations in simple polygons. In *Proceedings of the Fifth MSI-Stony Brook Workshop on Computational Geometry*, page 15, October 1995.

[10] Open GL Architecture Review Board. *OpenGL Reference Manual.* Addison-Wesley Publishing Company, Reading, MA, 1993.

[11] Open GL Architecture Review Board, J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide.* Addison-Wesley Publishing Company, Reading, MA, 1993.

[12] J. Plesńik. The NP-Completeness of the Hamiltonian Cycle Problem in Planar Digraphs with Degree Bound Two. *Information Processing Letters*, 8, no. 4:199–201, 1979.

[13] Silicon Graphics, Inc. Graphics Library Programming Guide, 1991.