Proceedings of ALGORITMY 2005
pp. 300–308

QUADRILATERAL MESHES STRIPIFICATION

PETR VANĚČEK*, RADEK SVITÁK§ , IVANA KOLINGEROVÁ†, AND VÁCLAV SKALA‡

CENTRE OF COMPUTER GRAPHICS AND DATA VISUALIZATION
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
UNIVERSITY OF WEST BOHEMIA
PILSEN, CZECH REPUBLIC
{PET,RSVITAK,KOLINGER,SKALA}@KIV.ZCU.CZ

Abstract. Quadrilateral models are becoming very popular in many applications of computer graphics (e.g. computer animation, computer games, scientific visualization of volumetric data, etc.). The complexity of these models can be very high (even millions of quadrilaterals), thus the problem of fast visualization of these meshes is often being solved. To increase the speed one can use some techniques to avoid sending of unnecessary faces (e.g., visibility culling) or some kind of simplification of complex objects (e.g., (C)LOD). Still it is important to reduce the time needed to transmit the set of faces by compressing the topological information and decompressing at the rendering stage.

One of popular approaches is to convert the quadrilateral mesh to the triangle mesh and render this mesh using strips of triangles. Using the strips, one can reduce the number of vertices sent to the rendering pipeline as two neighboring triangles in a strip share an edge and it is not necessary to send these vertices twice.

In this paper we present a new triangle stripping algorithm, designed for quadrilateral meshes. As this algorithm searches for a strip of quadrilaterals and splits them in the final stage it produces a high quality stripification.

Key words. computer graphics, visualization, quadrilaterals, triangles, triangle strips.

AMS subject classifications. 68R10, 68P30

1. Introduction.

Nowadays quadrilateral meshes are very often used to store and visualize various geometric objects in many applications such as computer games and movie industry (subdivision surfaces [13]), medical and scientific visualization (volume rendering, surface reconstruction from slices [9]), etc. In many of these applications a real time visualization is required. The speed of todays' high performance rendering engines is very often bounded by the rate at which the data is sent into the machine. Furthermore, most of the rendering engines can handle only triangle faces, thus the number of primitives increases.

To draw a set of i independent quads (quadrilaterals), we need to transmit $4i$ vertices. To reduce the amount of transmitted data, it is possible to split the quads into two triangles and connect them into triangle strips (or tristrips). In some graphic libraries a special type of primitives used for quads can be found (e.g., OpenGL). Rendering of quad strips is usually slower than rendering of triangle strips and the number of vertices is equal or higher than the number of vertices using tristrips (as we show next).

A *sequential tristrip* is a sequence of $j+4$ vertices that represents j quads: in Figure 1.1(a) the sequence (1,2,3,4,5,6) represents quads $\square 1243$ and $\square 3465$ (or triangles $\Delta 123$, $\Delta 324$, $\Delta 345$ and $\Delta 546$). A *sequential quad strip* is a sequence of $j+4$ vertices

*§This work was supported by Ministry of Education of The Czech Republic – project FRVŠ 1348/2004/G1

†‡This work was supported by Ministry of Education of The Czech Republic – project MSM 235200005

that represents j quads: in Figure 1.1(b) the sequence (1,2,3,4,5,6) represents quads □1243 and □3465.

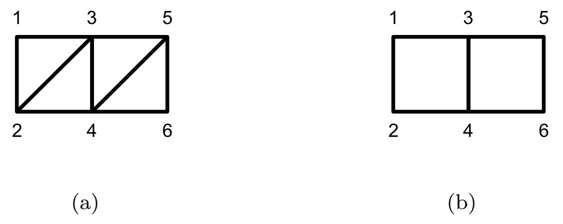


FIG. 1.1. *An example of sequential triangle strip 1.1(a) and a sequential quad strip 1.1(b).*

In some situations, the quad adjacency does not allow a sequential encoding. In Figure 1.2(a) the sequence (1,2,3,4,5,6,7,8) produces an invalid triangle Δ567. An extra vertex has to be added to change the sequence to (1,2,3,4,5,4,6,7,8). This operation is called *swap* and tristrips with swaps are called *generalized tristrips*. Using a quad strip, the situation is worse. In Figure 1.2(b) the sequence (1,2,3,4,5,6,7,8) produces an invalid quad □5687. To avoid this situation, it is necessary to make a *swap* at a cost of three additional vertices, i.e., a sequence (1,2,3,4,5,6,6,6,6,4,8,7).

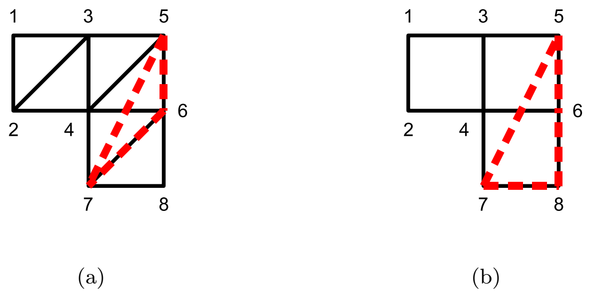


FIG. 1.2. *An example of a generalized triangle strip 1.2(a) and a generalized quad strip 1.2(b).*

From the above example it is obvious that triangle strips are more general and more efficient than quad strips. For this reason we concentrate on triangle strips in this paper. As the triangle strips can potentially reduce the amount of data transmission and transformation and lighting computations by a factor of three, many graphics libraries support it (e.g., Direct3D, OpenGL). Due to importance of this topic, many algorithms on stripification already exist. As the computing of an optimal set of tristrips is NP-complete [3], some heuristic is necessary. This means that each algorithm has its own advantages and disadvantages.

In this paper we describe a new stripification algorithm designed for quadrilateral meshes. In Section 2, some algorithms for quadrilateral mesh stripification are shown. Our new algorithm is presented in Section 3. Experiments and results are published in Section 4. Conclusion and future work are discussed in Section 5.

2. Existing Methods.

There are two possibilities how to construct triangle strips from not fully triangulated meshes. The first approach is to use some algorithm that triangulates the faces and then any stripification algorithm can be used. This way is general and can be used for any type of polygonal meshes. The main disadvantage of this approach is that it does not profit from the fact that the polygon can be triangulated arbitrarily. The other approach searches for strips in the untriangulated model and triangulates faces on the fly. Such an approach often leads to a better stripification.

Many works on constructing triangle strips were presented. Akeley et al. [1] have developed an algorithm, known as *tomesh* or *SGI* that converts a fully triangulated mesh into triangle strips. The algorithm tries to build tristrips which do not divide the remaining triangulation into too many small pieces. The strip is starting in the triangle with the least number of neighbors. Then a greedy heuristic is adding adjacent triangles with the least number of neighbors. If more triangles has the same number of neighbors, the algorithm looks one step ahead. There exists several modifications of this method using different heuristics [6, 11, 8, 5].

Xiang [12] developed an algorithm that can handle not fully triangulated meshes. This algorithm uses a dual graph based approach, i.e., it does not work with triangles but it uses a dual graph, where a node represents a triangle and neighboring triangles are connected with an edge in the graph. First, a stable algorithm for triangulation is used to triangulate non triangular faces. From this triangulation, the algorithm constructs a spanning tree in the dual graph of the triangulation. Then a dynamic procedure is used to partition the tree into a set of paths and greedily decomposes these paths into sequential strips. Finally, the strips are concatenated into longer ones. This algorithm produces a low number of vertices per triangle.

A method that takes the benefit of non triangulated data was developed by Evans [4] and it is known as *STRIPE.* It is free for non-commercial use and it is used by many authors for comparison. This algorithm is designed for meshes that are not fully triangulated and contain large number of quadrilateral faces. These faces are often arranged in large rectangular regions called *patches.* First, a *global* algorithm is used to analyze the mesh, find these patches and stripify them. For remaining faces a *local* SGI based algorithm is used.

Taubin [10] proposed an algorithm that can cover any connected manifold quadrilateral mesh without boundaries with a single strip. First, the algorithm finds an Eulerian circuit in the dual graph of the mesh. This circuit is partitioned to a set of Hamiltonian cycles. Then, these cycles are concatenated to a single strip by flipping a diagonal of the corresponding quads.

3. QStrip.

Nearly all stripification algorithms are designed for fully triangulated meshes or for meshes with arbitrary polygons, thus these methods cannot benefit from the regularity of quadrilateral meshes.

Our new algorithm (*QSTRIP*) is designed for meshes that are fully quadrilateral. It is based on a similar idea as the *SGI* algorithm for triangle meshes. As we are not working on a triangulated mesh, first we construct sequences of neighboring quadrilaterals. Then we traverse these sequences and triangulate the quadrilaterals such a way that each sequence can be covered with one triangle strip.

In the first step, the algorithm chooses a quadrilateral with a low number of neighbors to start a new strip. This choice minimizes the number of short strips. In Figure 3.1(a), the stripification process started in a quadrilateral with two neighbors and an isolated quadrilateral Q appeared. Usually we can avoid such a situation by starting from a quadrilatral with a low number of neighbors 3.1(b).

The chosen quadrilateral is removed from the mesh and it is inserted into the strip. The mesh is locally updated to reflect the quadrilateral removal. Now the algorithm chooses a neighboring quadrilateral that will be adjacent in the strip. To decrease

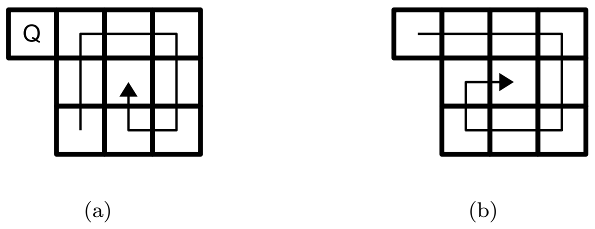


FIG. 3.1. *An example of a bad 3.1(a) and a good 3.1(b) choice of the starting quadrilateral.*

the number of vertices in the final stripification the algorithm preferentially chooses a quadrilateral that does not produce a swap (Figure 3.3(a)). The chosen quadrilateral is again removed from the mesh and inserted into the strip. These steps are repeated as long as it is possible (i.e., as long as there is a neighboring quadrilateral). If the mesh still contains some quadrilaterals, a new strip is started. A pseudo-code of this algorithm is presented in Figure 3.2.

```
while there is any quad in the mesh do
  start a new strip
  choose a quad with the lowest number of neighbors

  add the quad to the current strip
  remove the quad from the mesh
  locally update the mesh
  while there exists a neighbor of the current quad do
    choose a neighboring quad that does not produce a swap
    if such a quad does not exist then choose arbitrary

    add the quad to the current strip
    remove the quad from the mesh
    update the mesh
  end while
end while
```

FIG. 3.2. *Pseudo-code of the algorithm.*

The algorithm complexity is $O(s \cdot q + q)$, where q is the number of quads and s is number of strips in the final stripification, as we need $O(q)$ steps to find the starting quad for each strip. To speed up this algorithm, we use a priority queue for finding the quad with the lowest number of neighbors. Using such a structure decreases the complexity of finding the starting quad to $O(1)$, and the algorithm complexity is reduced to $O(s + q)$.

After the stripification phase, it is necessary to decompose the lists of quads into vertices of triangle strips. To provide a correct (counter-clockwise) orientation of triangle strips in the final mesh, it is necessary to start the first triangle of the strip in a counter-clockwise manner. This determines the diagonal of the first quad. As we cannot choose the first diagonal, three different situations can appear. In Figure 3.3(a) a sequential strip for four quads is shown. If the sequence of quads is not straight, a strip is preserved at a cost of one swap (Figure 3.3(b)) or two swaps (Figure 3.3(c)).

As the input meshes are fully 3D, in some cases it is not possible to split the quad arbitrarily, otherwise incorrect triangles appear. As there are four triangles incident

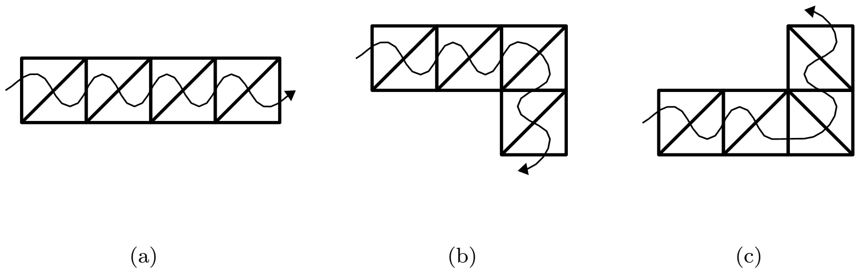

(a) (b) (c)

FIG. 3.3. *A straight sequence of quads can be covered by a sequential strip 3.3(a). To preserve a strip in a non-straight sequence of quads, it is necessary to use one swap 3.3(b) or two swaps 3.3(c).*

to one edge, the mesh is not manifold. Such a situation appears when two quads are neighboring through two edges (see Figure 3.4(a)). To avoid the incorrect triangles (Figure 3.4(b)), at least one of the quads has to be split along the diagonal that starts in the vertex that is not common for these two quads (Figure 3.4(c)). Respecting this criterion may lead to more swaps in the final stripification. Luckily this situation does not appear very often in a real life model.

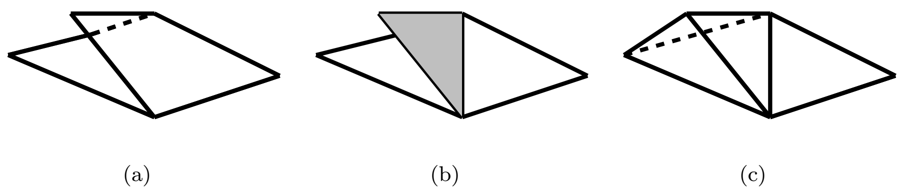

(a) (b) (c)

FIG. 3.4. *When two quads have two common edges 3.4(a), incorrect triangles may appear 3.4(b); the incorrect triangle is gray colored. To avoid it, at least one of the quads has to be split along the diagonal that starts in the vertex that is not common for these two quads 3.4(c).*

4. Experiments and Results. Our new algorithm has been implemented in Borland Delphi 7.0 as a part of a program for surface reconstruction from orthogonal slices (the reconstructed mesh is purely quadrilateral). The experiments were performed on a PC INTEL Pentium 4, 2.8 GHz, 2 GB of RAM, ATI FireGL T32 graphic card, running on MS Windows XP.

As our algorithm is designed specially for quad meshes, the quality of stripification is very high. We have compared our stripification algorithm with the *STRIPE* v.2 [2], which is also designed for quadrilateral meshes, and with the *FTSG* [12], which can handle non-triangulated meshes. Both algorithms were compiled with *gcc/cygwin* compiler.

A comparison of stripification methods is presented in Table 4.1. In the first two columns the number of vertices and the number of quads of the tested model are presented. In the next columns the number of strips and number of vertices (including swaps) obtained by tested algorithm are shown.

Our new method produces more or less the same number of vertices as *STRIPE*, but usually it covers the mesh by much smaller number of strips (especially for larger models). The *FTSG* method, which produces very good stripification for fully trian-

Table 4.1
Comparison of stripification methods. For each method the number of strips and the number of vertices in strips (including swaps) are presented.

		STRIPE		FTSG		QSTRIP	
vertices	quads	strips	vertices	strips	vertices	strips	vertices
2112	2114	88	5101	113	5675	4	4903
4000	4002	111	9517	258	10955	33	9516
8240	8236	140	17096	293	21549	8	17922
12592	12588	391	29050	664	33725	32	28290
16288	16290	393	36570	788	43333	37	36558
25712	25714	570	57181	1084	67931	49	57386
36264	36266	817	79957	1522	95801	44	80078
41919	42005	1356	98276	2405	112840	102	95998

gulated models, produces stripifications with very high number of vertices and strips in comparison to the *STRIPE* or the *QSTRIP*. The main reason for this big difference is that the *FTSG* makes a triangulation of the quadrilateral mesh first and then it stripifies the triangulated model. The *STRIPE* algorithm did not surprisingly create large patches but usually it created long sequential strips of quads (see Figure 4.1(a)). As these strips do not contain swaps, the number of vertices in the final stripification is comparable to our new algorithm although the *STRIPE* contains much higher number of strips. A visual comparison of the tested algorithms is presented in Figure 4.1. The tap model contains 16288 vertices and 16290 quads.

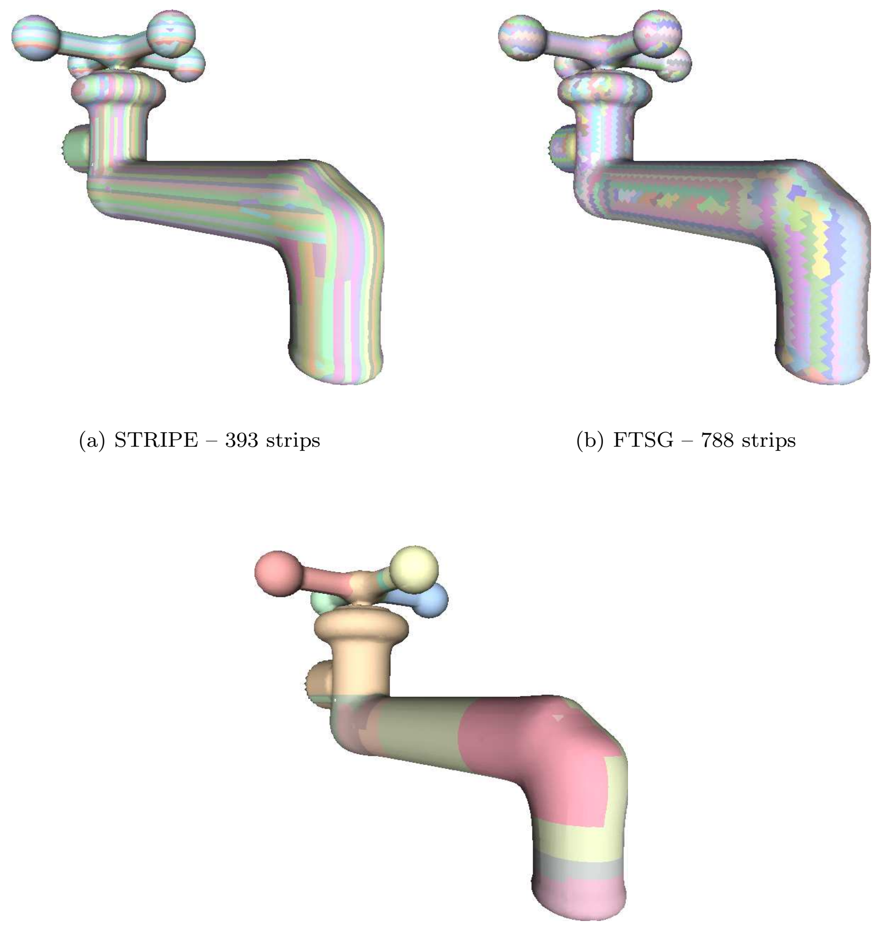
As the *STRIPE* algorithm outputs the stripification during the stripification process, it is not possible to exclude the time of I/O operations. For this reason we have included the time of I/O operations in all measurements, which can arise significant errors. To minimize these errors, all time measurements were performed five times and the minimal time is presented. Such a measurement can be a bit unfair to the *STRIPE* algorithm, as the write operation is not continuous, but on the other hand it is the real time that is needed for stripification. The comparison of running times is published in Table 4.2.

The running times of the *FTSG* are comparable to the *QSTRIP*. The difference in the running times can be partially caused by the cygwin emulation, as some functions have to be called from the cygwin dynamic library, but the main reason is probably the dynamic programming part of the *FTSG* algorithm.

The most time consuming step in the *STRIPE* algorithm is the global analysis which searches for the patches. As this global analysis searches the longest possible sequence of quads in both directions for each quad, it has $O(n^2)$ complexity for fully quadrilateral meshes.

In the last table (Table 4.3) we present the average frame rate (FPS) for models stripified by the tested methods and a ratio of frame rate for stripified models to frame rate for models rendered with quads. For the measurement we used OpenGL and vertex buffer objects (VBO) as they are preferred in new GPUs [7]. When using VBO, a sequential list of vertices is sent to the GPU (i.e., for each quad, four vertices are sent, for each triangle, three vertices are sent – 6 vertices for a quadrilateral face – and for each strip all vertices including swaps are sent).

As for a triangle mesh we have to send 1.5 times more vertices than for a quadrilateral mesh, the frame rate is much lower even though rendering a triangle primitive



(a) STRIPE – 393 strips

(b) FTSG – 788 strips

(c) QSTRIP – 37 strips

Fig. 4.1. *Visual comparison of stripification of a model of a tap (16290 vertices).*

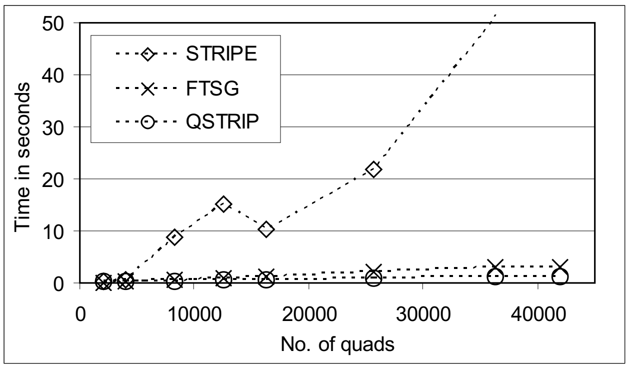
is faster than rendering a quadrilateral.

Although the number of vertices using triangle strips is nearly two times smaller than the number of vertices when using quads, the speed up is not two times higher. The reason is similar to the quad vs. triangle speed up (e.g. drawing the triangle strip primitive is more time consuming than drawing the quad).

The comparison of frame rates of individual stripification methods did not get any surprising results. The *FTSG* produces a stripification that is rendered at the lowest frame rate as it contains high number of vertices and strips. The stripification produced by our algorithm runs at highest frame rate as the number of vertices and strips is low. Although *STRIPE* produces a stripification of nearly the same number of vertices as *QSTRIP*, the frame rate is in the middle between *QSTRIP* and *FTSG*. This is caused by the fact that *STRIPE* stripification contains higher number of strips and starting a new triangle strip costs some extra time.

Table 4.2
Comparison of running times (in seconds). For each method the running time (including I/O operations) is presented.

vertices	quads	STRIPE	FTSG	QSTRIP
2112	2114	0.31	0.14	0.25
4000	4002	0.51	0.25	0.30
8240	8236	8.69	0.70	0.39
12592	12588	15.18	0.97	0.51
16288	16290	10.20	1.31	0.60
25712	25714	21.79	2.06	0.84
36264	36266	50.81	2.89	1.11
41919	42005	83.58	3.10	1.28



In all these tests our new algorithm reached the best results. These tests are a bit unfair to *STRIPE* and *FTSG* as these algorithms can handle more general type of meshes, on the other side as far as we know, there is no other algorithm designed for fully quadrilateral meshes, thus we have chosen the best existing algorithms.

5. Conclusion. We have designed and implemented a new stripification algorithm for quadrilateral meshes. As we know the mesh structure, we can exploit it and produce a high quality stripification. In comparison to other methods that can stripify not fully triangulated meshes, our new algorithm produces a stripification with lower number of strips and vertices.

In the future work we would like to improve the quality of stripification (especially to decrease the number of vertices). We also would like to investigate the behavior of vertex caches that are implemented in todays GPUs and adapt the stripification to maximize the benefit of the cache.

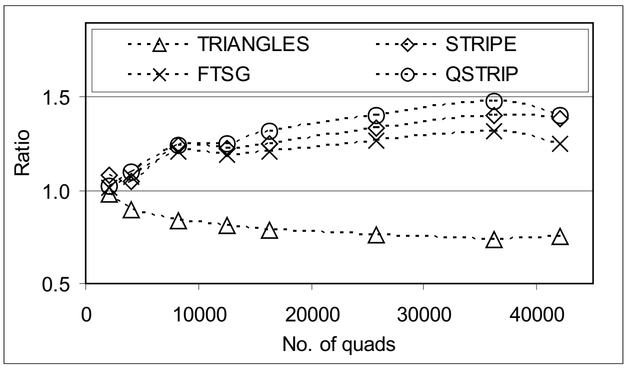
REFERENCES

[1] K. Akeley, P. Haeberli, and D. Burns. tomesh.c. C Program on SGI Develope's Toolbox CD, 1990.
[2] F. Evans. Stripe, 1998. http://www.cs.sunysb.edu/ ˜stripe/.
[3] F. Evans, S. Skiena, and A. Varshney. Completing Sequential Triangulations is Hard. Technical

Table 4.3

Comparison of frame rate achieved with models rendered with quads, triangles and tested stripifications.

no. of	QUADS		TRIS		STRIPE		FTSG		QSTRIP	
quads	FPS	ratio	FPS	ratio	FPS	ratio	FPS	ratio	FPS	ratio
2114	234	1.00	229	0.98	252	1.08	238	1.02	240	1.03
4002	291	1.00	261	0.90	305	1.05	312	1.07	320	1.10
8236	229	1.00	192	0.84	285	1.25	276	1.21	283	1.24
12588	180	1.00	147	0.82	220	1.22	214	1.19	224	1.25
16290	166	1.00	130	0.79	207	1.25	200	1.21	218	1.31
25714	125	1.00	95	0.76	167	1.33	159	1.27	176	1.40
36266	99	1.00	73	0.74	139	1.40	130	1.32	146	1.48
42005	80	1.00	60	0.75	111	1.39	100	1.25	111	1.40



report, Department of Computer Science, State University of New York at Stony Brook, 1996.

[4] F. Evans, S. Skiena, and A. Varshney. Optimizing Triangle Strips for Fast Rendering. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pp. 319–326, 1996.

[5] O.M.van Kaick, M.V.G.da Silva, and H. Pedrini. Efficient Generation of Triangle Strips from Triangulated Meshes. In *Proceedings of the 12th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG'2004)*, pp. 475, 2004.

[6] D. Kornmann. Fast and Simple Triangle Strip Generation. Technical report, VMS Finland, Espoo, Finland, 1999.

[7] NVIDIA Corporation. Using vertex buffer objects. White Paper: http:// developer.nvidia.com/ object/ using_VBOs.html, 2003.

[8] M.V.G.da Silva, O.M.van Kaick, and H. Pedrini. Fast Mesh Rendering through Efficient Triangle Strip Generation. In *Proceedings of the 10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG'2002)*, pp. 127–134, 2002.

[9] R. Sviták and V. Skala. Robust Surface Reconstruction from Orthogonal Slices. In *Electronic Computers and Informatics (ECI'04)*, pp. 451–456, 2004.

[10] G. Taubin. Constructing Hamiltonian Triangle Strips on Quadrilateral Meshes. In *International Workshop on Visualization and Mathematics 2002*, 2002.

[11] P. Vaněček. Comparison of Stripification Techniques. In *6-th Central European Seminar on Computer Graphics CESCG'02*, pp. 65–74, 2002.

[12] X. Xiang, M. Held, and P. Mitchell. Fast and Effective Stripification of Polygonal Surface Models. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1999.

[13] D. Zorin and P. Schröder. Subdivision for Modeling and Animation. Technical report, SIGGRAPH 2000, 2000. course notes.