Comparison of Stripification Techniques

Petr Vaněček[1]
pet@students.zcu.cz

Centre of Computer Graphics and Data Visualization
Department of Computer Science and Engineering
Pilsen, Czech Rep.

Abstract

Triangle surface models are nowadays most often types of geometric objects description in computer graphics. Therefore, the problem of fast visualization of this type of data is often solved. One of popular approaches is stripification, i.e., a conversion of triangle surface into strips of triangles. This enables to reduce the rendering time by reduction data size and by avoiding of redundant lighting and transformations computations.

This paper describes the comparison results of classic STRIPE and greedy SGI based algorithms for stripification.

KEYWORDS: strip, triangle, mesh compression, OpenGL, computer graphics.

1 Introduction

In various computer graphics applications such as CAD, digital cartography, medicine, etc., triangle models are often used to visualize surfaces and volumes. The speed of high performance rendering engines on triangular meshes is usually bounded by the rate at which triangulation data is sent into the machine. To draw independent triangles we need to transmit three vertices per triangle to the graphic system, but we can minimize the data by an ordering the triangles so that consecutive triangles share an edge. Using such an ordering, called *triangle strip*, the first triangle is defined by three vertices and the following triangles by only one additional vertex. This reduces the rendering time by avoiding redundant data transmission to engine and redundant lighting and transformation computations. As the datasets in practice are usually huge, substantial speedup and/or data compression [8] may be achieved in this way.

The triangle strip primitive is supported in many graphics libraries such as Irix-GL or OpenGL. In Fig. 1 a *sequential triangle strip* is shown. In this case each triangle is described by ith, $(i+1)$st, and $(i+2)$nd vertices of the strip. Using the

[1]This work was supported by the Ministry of Education of The Czech Republic - project MSM 235200005.

sequential strip, the transmit cost of n triangles can be reduced from $3n$ to $n + 2$ vertices.

There also exists a situation where the triangle adjacency does not allow a sequential encoding. In Fig. 2 we have to break the strip, add an inverted edge $4', 4$ and continue with the rest of sequence. This inverting operation is called *swap* and strips using these swaps are called *generalized strips.*

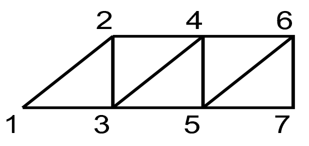

Figure 1: The order of vertices to send is 1,2,3,4,5,6,7.

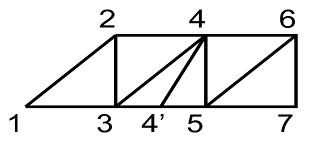

Figure 2: The order of vertices to send is 1,2,3,4,4',swap/4,5,6,7.

The swap had been implemented as a one-bit attribute in GL. For better portability it was decided to not support it in OpenGL and one more vertex is sent to the engine instead. This means that an empty triangle appears. Although the swap costs one vertex, it is still cheaper to use the swap instead of starting a new triangle strip that costs two vertices.

Due to importance of this topic, many algorithms already exist. Because the computing of an optimal set of triangle strips is NP-complete [5], some heuristic is necessary. This means that each method has its own advantages and disadvantages. In this paper we want to explore and compare some of the existing algorithms. Using this knowledge, we want to develop our own stripification algorithm in the future.

There exist several different criteria of strip quality, such as number of strips, number of vertices, locality of the strip etc. The locality criteria are mainly useful if level-of-detail clipping of geometric models in complicated scenes is expected. As, at present, we solve only static visualization of simple scenes, we concentrated on number of strips and number of vertices criteria.

In Section 2, some of existing methods for non-hierarchical meshes are shown. In section 3, we describe the SGI based algorithms into more details [1]. Another OpenGL speed-up technique that can be combined with strips is presented in section 4. Experimental results are published in section 5. Conclusion and future work are discussed in section 6.

2 State of the Art

Many works on constructing triangle strips were presented. Arkin et al. [4] show that testing whether a given triangulation of point set or polygon is Hamiltonian (i.e. the dual graph of triangulation contains a Hamilton cycle) can be done in linear time. They prove that related problem of computing a Hamilton triangulation is NP-complete for polygon with holes.

Akeley et al. [1] have developed an algorithm, known as *tomesh* or *SGI algorithm*, that converts a fully triangulated mesh into triangle strips. The algorithm tries to build triangle strips which do not divide the remaining triangulation into too many small parts. The strip is starting in the triangle with the least number of neighbors. Then a greedy heuristic is adding neighboring triangles with the least number of neighbors. If more triangles has the same number of neighbors, the algorithm looks one step ahead. As this algorithm is one of the most often used, we decided to pick it for implementation. Therefore, more details are presented in Section 3.

Other method for stripification is STRIPE [6, 7]. It is public free and it is used by many authors for comparison. We will also use it as one of method, to be able to compare our future algorithm to some other (indirectly). The main idea of this algorithm is base on a fact that there exists a lot of models that are not fully triangulized (i.e., contains quadrilateral faces). These faces are often arranged in large rectangular regions called "patches". In the first pass (called "Global algorithm") the algorithm analyze these patches and stripify them. For remaining triangles a "Local algorithm" is used. The local algorithm is based on the same idea as the SGI algorithm.

A method based on a spanning tree duality is presented in [12]. This algorithm constructs a spanning tree in the dual graph of the triangulation. Then it partitions the tree into a set of paths, corresponding to Hamiltonian triangulations, and greedily decomposes the corresponding Hamiltonian strips into sequential strips. Finally it concatenates short strips into longer strips using a set of postprocessing heuristics.

The tunneling algorithm [11] can be used for both static and continuous level–of–detail (CLOD) meshes. It uses the dual graph representation. There are two different edges in this representation – "strip edge" that is a part of strip and "non-strip edge". The tunnel in the dual graph is a sequence joining two ends of strips and alternating between strip and non-strip edges. By changing the strip edge for non-strip edge and vice versa, the number of strips is reduced by one.

3 SGI-based Algorithm Implementation

In this section some details about SGI-based stripification algorithm will be explained. The SGI *tomesh* was developed for generic triangle strips. Because it was designed for GL which supports a swap command, it doesn't try to avoid the swaps. This is no problem for the GL library because you need to send only one bit per swap. But for OpenGL library it is one vertex per swap. This means that the generalized strip containing n triangles could be encoded with more than $n+2$ vertices.

3.1 Basic Algorithm

The basic algorithm consists of several steps:

1. If there are no more triangles in the triangulation then exit.
2. Find the triangle t with the least number of neighbors (if more than one exists, choose arbitrary).
3. Start a new strip.
4. Insert the triangle t to the strip and remove it from the triangulation.
5. If there is no neighboring triangle to triangle t then go to 1.
6. Choose a new triangle *t'*, neighboring to triangle *t*, with the least number of neighbors. If there is more than one triangle *t'* with the same least number of neighbors, look one level ahead. If there is a tie again, choose *t'* arbitrarily.
7. $t \leftarrow t'$. Go to 4.

The effectiveness of the second step is crucial for the algorithm complexity. The best possible complexity is a linear function of the number of triangles n. If the first step is implemented as a sequential search then the algorithm complexity is close to $O(n \cdot s)$, where s is the number of triangle strips. To avoid this growth of complexity, some extra-structures have to be used (lists, priority qeue, etc.).

3.2 Choosing next triangle

Because step 6 – i.e., choosing next triangle – is a heuristic and has the most significant impact on the strip quality, we decided to try some other heuristic functions and study the influence on the results. We have tested these heuristics:

1. Choose the next triangle randomly (thereinafter RN).
2. Choose the triangle with the least number of neighbors.
   If more than one exists, choose arbitrary (thereinafter LNRN).
3. Choose the triangle with the least number of neighbors.
   If more than one exists then look one level ahead (original heuristic - thereinafter LNLN).
4. Choose the triangle with the least number of neighbors.
   If more than one exists then choose the one that does not produce a swap (thereinafter LNLS).

The first heuristic is very easy to implement. It should be fast, but it does not reflect the main idea of the algorithm – not to break the existing triangulation into many small parts. The second heuristic is more complex than the first one, but still more simple than the original one. The result should be better (less fragmented triangulation) than in the first case. The last heuristic is about the same complexity as the original (third) one. Instead of trying to look ahead when a tie occurs, a test for a swap is made. This heuristic should give more strips than the original one, but probably less vertices.

4 OpenGL Speedup

While using the OpenGL library, some other speed-up improvements could be done. Few years ago, new extensions had been added to OpenGL system. These extensions are working with vertex arrays and indices to these arrays. If not using these extensions, three vertices (3*3 coordinates) are needed to define each triangle. For two triangles 3*3*2 coordinates are needed, etc. In fact, the vertex coordinates are shared by more than one triangle (e.g.,in Delaunay triangulation one vertex is shared by approximately 6 triangles). The reduced data size should be twice smaller than the original size. Original size is $t * 3\ vertices * 3\ coordinates = 9 * t$, while reduced size is $v * 3\ coordinates + t * 3\ indices$, where t is the number of triangles and v is the number of vertices. Because in a usual triangle model the number of vertices is about twice smaller than the number of triangles, the reduced size is $\frac{t}{2} * 3 + t * 3 = \frac{9}{2}t$.

It's also possible to use these extensions in combination with triangle strips. The reduced data size while using triangle strips and OpenGL extension should be about 6 times smaller than the classic method (three times smaller using triangle strips and twice smaller using the extension).

5 Experiments and Results

All versions of the algorithm from section 3 were implemented in Borland Delphi 5.0. It has been tested on 25 models from data sets from [9, 10, 2]. Experiments have been performed on a PC AMD Duron 850MHz with 256MB of RAM and Geforce 2 MX graphic card, running on MS Windows 2000 system. The implementation was compared to one of the best known publicly available algorithm - STRIPE (http://www.cs.sunysb.edu/˜stripe/) with default settings. The times of I/O operations have been excluded from measurements.

Tables 1 and 2 show comparison of the classic OpenGL rendering method, extensions, triangle strips and triangle strips combined with extensions. Table 1 shows the frame rate and Table 2 shows the ratio of tested method frame rate to classic method frame rate. To be able to make one triangle strip per mesh even for large data sets, we decided to use artificial data (i.e., planar rectangular grid). In these two tables we want to show the advantage of using triangle strips and OpenGL extensions.

# of triangles	5k	19k	44k	79k	124k	179k	244k	318k	403k
classic	138	56.2	28.7	18.7	12	8.5	6.2	5.2	4.2
extension	124	97.7	53.5	33	21	15.5	11.5	9.7	7.7
strips	111	75.5	72.5	45.5	30.5	22.7	17	14.2	11.5
extension + strips	99.5	142	82.5	66.7	45.2	39.7	31.2	26.3	22

Table 1: Frame rate - comparison of classic calls, extensions, triangle strips and extensions + triangle strips. All values are in frames per second.

# of triangles	5k	19k	44k	79k	124k	179k	244k	318k	403k
classic	1	1	1	1	1	1	1	1	1
extension	0.90	1.74	1.86	1.76	1.75	1.82	1.86	1.86	1.84
strips	0.80	1.34	2.53	2.43	2.54	2.67	2.76	2.73	2.76
extension + strips	0.72	2.53	2.87	3.57	3.77	4.67	5.06	5.06	5.28

Table 2: Ratio - comparison of classic calls, extensions, triangle strips and extensions + triangle strips. Values are computed relatively to the classic method.

Extension speedup is close to theoretical bounds (about 1.85 for larger data sets). The difference is probably caused by the border of the grid, where the vertices are used only three times. The speedup while using triangle strip is a bit smaller (about 2.7 for larger data sets) than expected. This difference could be caused by the swaps that are needed to cover the grid. While using the combination of both methods, the speedup is close to the theoretical expectation (strips speedup * extension speedup). This measurement also affirms the theory that the speed is not bounded by the rendering process itself but by the rate at which the data are transmitted into the graphical processor.

Table 3 shows the information about data sets used in tests presented below. We have chosen eleven models that are often used in other papers and are available.

#	model	# vertices	# polygons	#	model	# vertices	# polygons
1	sphere	146	288	7	bunny	35947	69451
2	f2	961	1800	8	bell	213373	426572
3	elipsoid	2452	4900	9	hand	327323	654666
4	cow	2905	5804	10	dragon	437645	871414
5	demi	9138	17506	11	happy buddha	543652	1087716
6	teeth	29166	58328				

Table 3: Set of testing models.

Next tables show a comparison of the SGI based method with all four modifications and the STRIPE algorithm. Table 4 shows the number of strips in the models. For bigger models, the behavior of all modifications of SGI is the same as we expected. RN method is the worst one and LNLN gives the best results. Except the *ellipsoid (3)*, STRIPE is worse than the SGI LNLN method. For bigger data the STRIPE algorithm was too slow and we got incorrect output (the same behavior as reported e.g. in [11]).

model	1	2	3	4	5	6	7	8	9	10	11
method											
RN	17	90	26	185	812	4083	2406	18103	38259	62911	78192
LNRN	2	22	3	110	412	1319	927	10578	10983	20617	25685
LNLN	1	15	24	98	336	1068	648	8094	9381	17401	21555
LNLS	1	13	48	158	452	1776	1291	11680	15780	26639	33187
Stripe	2	18	1	101	385	1255	917				

Table 4: The number of strips needed for model.

In Table 5, the number of vertices after stripification is shown (for graphic interpretation see Fig. 3). The LNLS method is better than the original method. STRIPE algorithm is worse than LNLN and LNLS method (except ellipsoid (3)).

model	1	2	3	4	5	6	7	8	9	10	11
method											
RN	366	2.8k	6.3k	7.7k	24k	89k	92k	598k	1.0M	1.3M	1.7M
LNRN	363	2.6k	5.3k	7.7k	23k	85k	92k	591k	944k	1.3M	1.6M
LNLN	335	2.5k	5.1k	7.6k	23k	82k	86k	584k	886k	1.2M	1.5M
LNLS	313	2.4k	5.1k	7.1k	22k	76k	82k	543k	823k	1.1M	1.4M
Stripe	363	2.5k	5.0k	7.6k	23k	84k	91k				

Table 5: The number of vertices needed for model.

The comparison of time is presented in Table 6 (for graphic interpretation see Fig. 4). Although the condition in RN method is short, the time is worse than the other methods. This is probably caused by bigger number of strips and bigger number of vertices which increase the number of other decisions. The fastest is method LNRN. The STRIPE algorithm seems to be very slow (four and more times slower than any other method).

model	1	2	3	4	5	6	7	8	9	10	11
RN	0	20	41	40	120	511	581	3385	4967	7851	9915
LNRN	0	10	30	50	130	450	591	3375	4697	6960	7992
LNLN	0	20	30	40	130	471	571	3435	4787	7080	8002
LNLS	0	10	40	40	131	461	581	3475	4917	7281	8312
Stripe	0	120	7621	220	551	1963	3585				

Table 6: Time needed for stripification in milliseconds.

One of the reasons why the STRIPE seems to be so bad is that STRIPE is constructed for data sets that are not fully triangulized, but our testing models are fully triangulized. This is a bit unfair, because only the *local algorithm* is used. The reason why we decided to use STRIPE as a testing method is that many other authors use it and so we are able to compare our method to them indirectly. We do not know why the STRIPE fails on bigger data, but it is not only our experience. The same behaviour is mentioned also in other papers ([11, 3] and even the authors of STRIPE [6] are testing the algorithm on small data only). We briefly checked the code of STRIPE and we found some non-optimal parts of code that slow STRIPE down.

We have also tested the length of strips. The typical histogram obtained by LNLN method is shown on Fig. 5 (dragon). As you can see, in a typical model most of the strips is quite short but there exists a limited number of strips that are much longer. We have tried to normalize the histogram by cutting the long strips to avoid breaking the triangulation, but the effect was different than we expected: both the number of triangles and the number of vertices increased as we have decreased the maximum length of a strip.

From these results, a conclusion can be done that from SGI-based method, LNLN provides the smallest number of strips while LNLS needs the smallest number of vertices. The difference in rendering time while using a model stripified by LNLN and LNLS method is less than 1%. RN and LNRN modification provided slightly worse results. The STRIPE algorithm gave relatively bad results although it is widely used as a comparative standard in stripification papers.

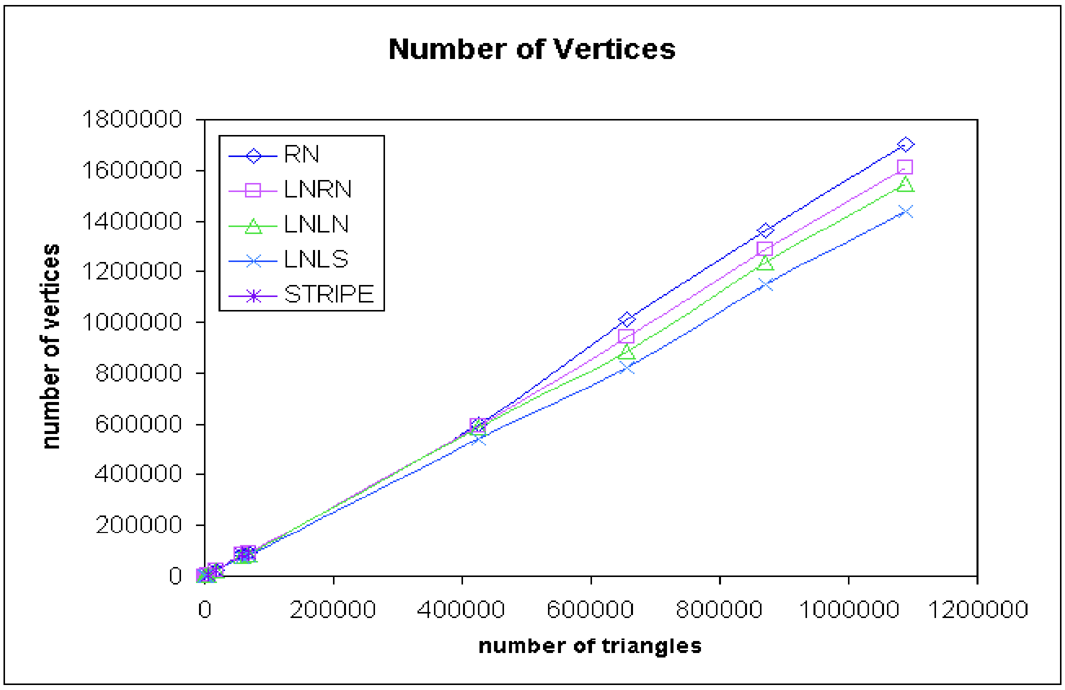

Figure 3: Number of Vertices.

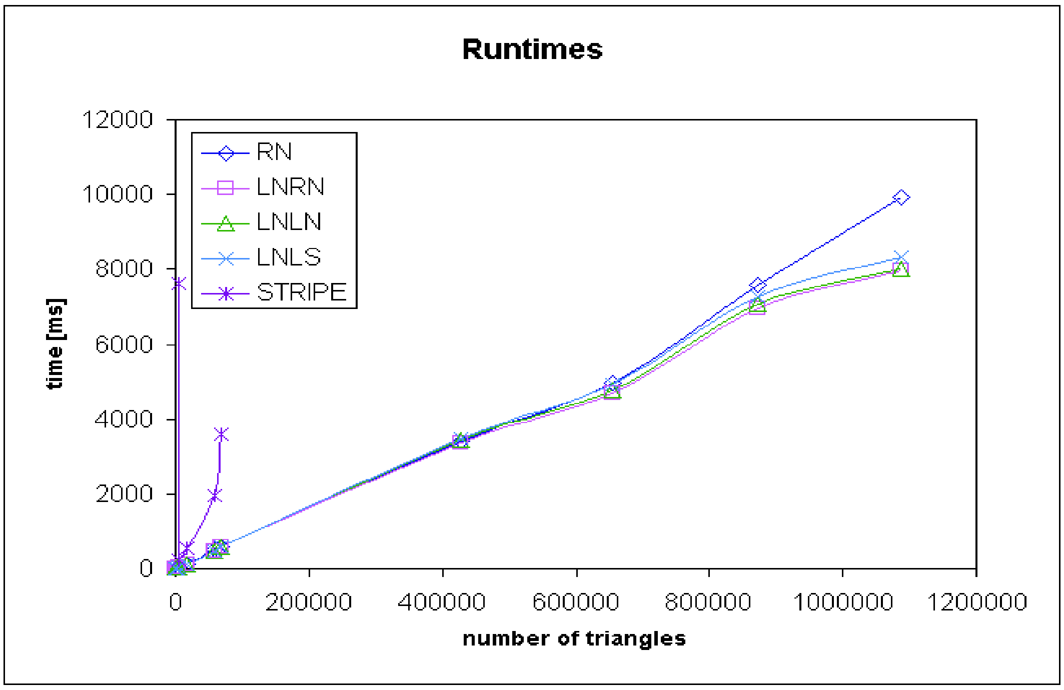

Figure 4: Runtimes.

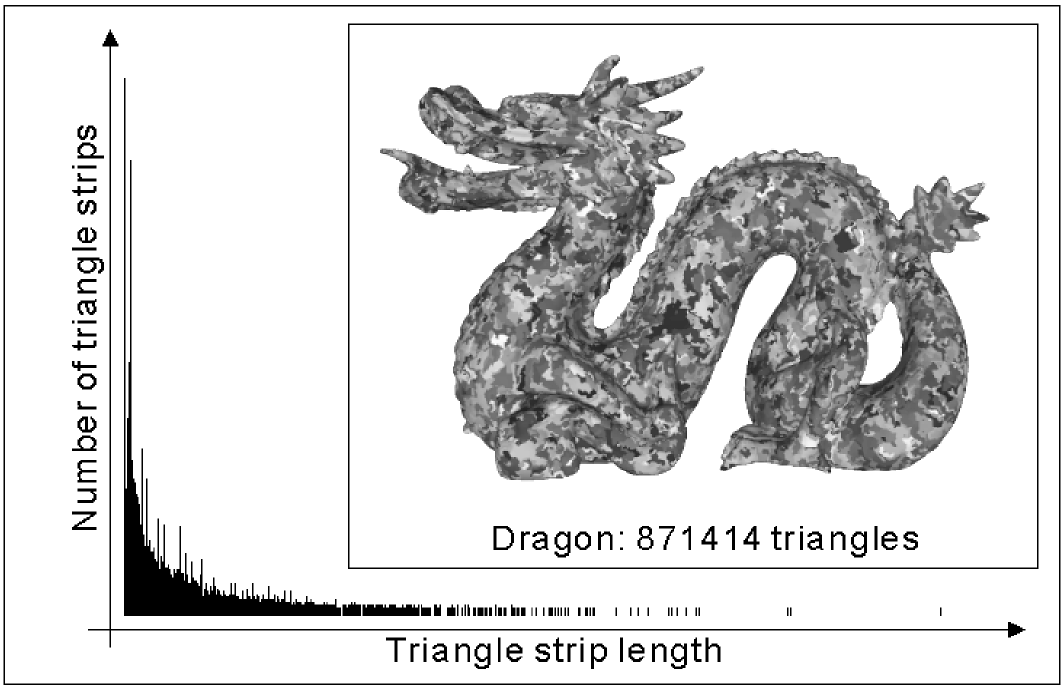

Figure 5: Typical histogram of length of strips using LNLN method.
The Dragon model has 17401 strips.

6 Conclusion

We have implemented several versions of the so called SGI method and compared them to an existing and often used algorithm - STRIPE. We wanted to get some ideas about the stripification – whether it is useful, what is the typical behaviour, etc. We will use this experience in the future as a theoretical base for our new intended stripification algorithm.

References

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

[2] University of West Bohemia CCGDV. Data archive. http://herakles.zcu.cz/research/mve/download.php.

[3] M.V.G. da Silva, O.M. van Kaick, and H. Pedrini. Fast mesh rendering through efficient triangle strip generation. In *WSCG'2002*, pages 127–134, 2 2002.

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

[5] F. Evans, S. Skiena, and A. Varshney. Completing sequential triangulations is hard. Technical report, Department of Computer Science, State University of New York at Stony Brook, 1996.

[6] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 319–326, 1996.

[7] F.Evans. Stripe, 1998. http://www.cs.sunysb.edu/˜stripe/.

[8] Martin Isenburg. Triangle strip compression. Technical Report TR00-003, University of North Carolina at Chapel Hill, 2000.

[9] Stanford Computer Graphics Laboratory. http://graphics.stanford.edu/data/3Dscanrep/.

[10] Georgia Institute of Technology. Large geometric models archive. http://www.cc.gatech.edu/projects/large_models/.

[11] A. James Stewart. Tunneling for triangle strips in continuous level-of-detail meshes. In *Graphics Interface*, pages 91–100, June 2001.

[12] Xiang, Held, and Mitchell. Fast and effective stripification of polygonal surface models (short). In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, pages 985–986, 1999.