Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes

Alexander Bogomjakov Craig Gotsman

Computer Science Department
Technion – Israel Institute of Technology
Haifa 32000, Israel

{alexb|gotsman}@cs.technion.ac.il

Abstract

We present methods to generate rendering sequences for triangle meshes which preserve mesh locality as much as possible. This is useful for maximizing vertex reuse when rendering the mesh using a FIFO vertex buffer, such as those available in modern 3D graphics hardware. The sequences are universal in the sense that they perform well for all sizes of vertex buffers, and generalize to progressive meshes. This has been verified experimentally.

Key words: Rendering sequence, transparent vertex caching, triangle strips, progressive meshes, space-filling curves.

1 Introduction and Previous Work

One of the trends in contemporary computer graphics applications is the use of more and more polygons in order to increase image realism. This trend is partially fuelled by recent developments in graphics hardware, particularly the appearance of the GPU (Graphics Processing Unit) on low-end display adaptors. This means that not only the scan conversion is done by the graphics adaptor, but also the 3D geometric projections and shading operations. Hence, processing of the scene geometry is no longer a bottleneck as it was in the past.

In order to process geometry as rapidly as possible, the GPU's (e.g. the NVidia GeForce 1 and 2) maintain a FIFO vertex cache of fixed size, thru which processed vertices travel. While rendering a typical 3D mesh on a per-triangle basis, each vertex may have to be processed more than once, since each vertex participates in six triangles on the average. Processing a cached vertex can be significantly faster than processing an uncached vertex. Thus, to maximize benefit from the cache, the mesh triangles, hence also the associated vertices, must be rendered in an order which somehow preserves locality. This ordering of the triangles is called the mesh *rendering sequence.* A good rendering sequence will minimize the average number of cache misses per triangle, also known as the ACMR (average cache miss ratio) which, theoretically, is anywhere between 0.5 and 3.0, since the number of triangles in a typical 3D mesh is approximately double the number of vertices. Fig. 1 shows a mesh and a possible rendering sequence. Note that the sequence is not necessarily continuous, i.e. triangles adjacent in the rendering sequence are not necessarily adjacent in the mesh.

3D meshes are usually specified, for example, in the ASCII VRML 2.0 file format, as a list of triangles in an arbitrary order, where each triangle is specified as three indices into a list of vertices. Simple-minded renderers send these triangles to the graphics pipeline in the order specified in the file, hence achieve mediocre rendering performance. More sophisticated renderers use the *triangle strips* technique, which renders the triangle mesh using a FIFO vertex cache of size 2, which is a standard part of legacy 3D hardware. Algorithms to generate triangle strips were described by Akeley et al [1], Evans et al [8], Xiang et al [24] and Stewart [21]. However, due to the limited size of the cache, it is provably not possible to reduce the ACMR below 1.0 in this case. Deering [6] first proposed a hardware model where a larger vertex cache is allowed, which he called *generalized triangle meshes*, but did not supply algorithms to generate the appropriate rendering sequences. Chow [5] later provided algorithms, as did Bar-Yehuda and Gotsman [4] and Lin and Yu [15]. Deering's hardware design has since been implemented in Sun Microsystems Elite3D graphics hardware series [22] and the generalized mesh representation is an important component of the compressed geometry format of the Java3D API [22]. Recently, Mitra and Chiueh [16] also proposed an architecture based on two vertex buffers and a breadth-first traversal algorithm for generating rendering sequences for it.

The published algorithms, however, used a non-FIFO vertex cache, i.e. a cache which could be explicitly controlled by the user, which is non-existent in today's general-purpose low-end GPU's. Realizing this, Hoppe [12] proposed an algorithm to generate rendering se-

quences for a so-called *transparent* FIFO cache, and experimentally showed that for any given cache size, his algorithm generates rendering sequences whose ACMR is not significantly worse than those generated by Chow's algorithm. However, a major problem with all the algorithms, including Hoppe's, is that the cache size must be known in advance, i.e. a *different* rendering sequence is generated for any cache size, and using it to render a mesh when a smaller cache is present may provide far from optimal results.

Another drawback of all the existing algorithms is that they do not generalize well for *progressive* meshes. Progressive meshes differ from fixed-resolution meshes in that a vertex removal order is imposed on them, usually for reasons of geometric approximation. As each vertex is removed in turn from the mesh, the resulting *hole* is retriangulated. Progressive meshes are useful in rendering a mesh at a resolution which can continuously vary depending, say, on viewing parameters. A sequence of *update records* would indicate the vertex removals and retriangulations to perform in order to achieve the desired polygon count. Since the mesh is constantly changing, the rendering sequence must also change with it. The only work we are aware of in this respect is that of El-Sana et al [7] and Stewart [21] on maintaining simple triangle strips for progressive meshes.

This paper introduces methods for generating *universal* mesh rendering sequences. These sequences preserve locality at all scales, hence may be used as rendering sequences with a FIFO cache of any size. We also show how, thanks to the universality of the sequences, these rendering sequences may be adapted to progressive meshes without a significant performance penalty. In a sense, the rendering sequences generated by our algorithms are analogous to the so-called *discrete space-filling curves* [18], highly regular constructions applicable only to uniform grid structures. The simplest application of these sequences, once computed, is to list a triangle mesh in the order dictated by the sequence in an ASCII VRML file, so even the simple renderers may benefit from them.

Space-filling curves are classics dating back to Hilbert, Peano and other mathematicians. See the book by Sagan [18] for a complete treatise on the subject. These curves are actually traversals of the cells in a (multi-dimensional) grid, which preserve locality in some sense. Quantification of the notion of locality-preservation has also been the focus of recent attention, and a variety of measures have been proposed (e.g. [9,17]), tailored to specific applications. The essence of

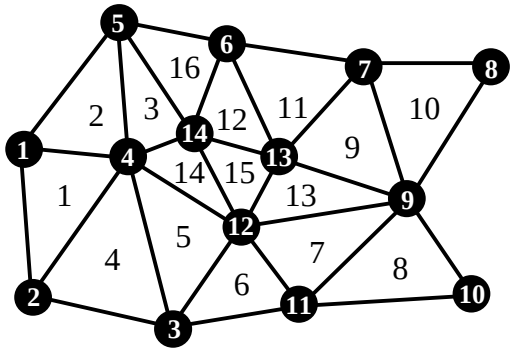


Figure 1: A triangle mesh containing 14 vertices and 16 triangles and a possible rendering sequence. Triangles are numbered in rendering sequence order. The rendering sequence is <1,2,4>, <1,4,5>, <4,5,14>, <2,3,4>, <3,4,12>, <3,11,12>, <9,11,12>, <9,10,11>, <7,9,13>, <7,8,9>, <6,7,13>, <6,13,14>, <9,12,13>, <4,12,14>, <12,13,14>, <5,6,14>. There are 25 cache misses (ACMR = 1.56) for a cache of size 4 and 14 cache misses (the minimum, ACMR = 0.88) for a cache of size 16.

our work is to generalize this notion to general irregular triangle meshes, where the classical methods fail. Attempts at such constructions have been made by Bartholdi and Goldsman [2], but the effectiveness of the construction was not quantified in their work. Note that these traversals depend only on the *connectivity* of the mesh, and not on its geometry, i.e. the coordinates in space of the vertices.

It might be argued that the precise connectivity of a mesh is just an artifact of the specific method used to create the mesh, hence it would be reasonable to allow modification of the connectivity in order to generate good rendering sequences more easily, as long as the geometric shape of the mesh is preserved. Some applications even perform *remeshing*, which modifies the number and the geometry of the mesh vertices in order to achieve a more regular connectivity. While this is true in some applications, we make the more stringent assumption that the connectivity cannot be changed at all.

It might also be argued that if the algorithm computing the rendering sequence is fast enough, it could be run on the fly immediately before rendering, eliminating the need for a precomputed *universal* rendering sequence suitable for all cache sizes, since at this point the vertex cache size of the rendering hardware is known, and a rendering sequence tailored to the cache size (such as Hoppe's [12]) can be used. This might be true in theory, but in a typical client-server scenario, where the server is very powerful, and the client very weak (e.g. a PDA), it is very important to reduce the compute load on the client to a minimum, hence a universal rendering se-

quence precomputed and stored at the server, is advantageous.

From a theoretical point of view, Bar-Yehuda and Gotsman [4] have shown that a vertex cache of size $\theta(\sqrt{n})$ is required in order to render a *n*-vertex triangle mesh with the minimum ACMR of 0.5. Conversely, given a cache of size k, they show that the ACMR is $0.5+\Omega(1/k)$. These bounds apply to the case of a controllable cache, so a (more restricted) FIFO cache can perform no better. In general, the objective is to generate a rendering sequence, such that when each triangle is rendered, hopefully as many of the triangle vertices as possible will be present in the cache. If not – these count as cache misses.

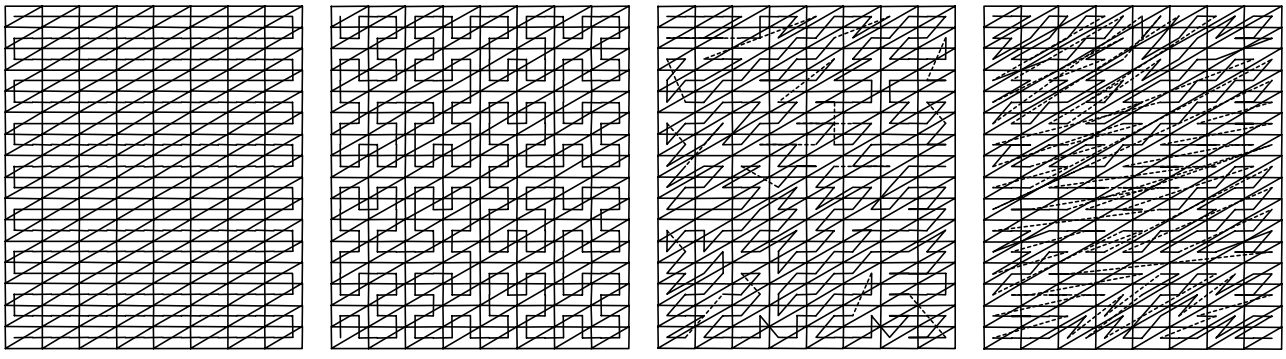
The remainder of this paper is organized as follows. Section 2 presents two algorithms for generating universal rendering sequences, which are generalized to progressive meshes in Section 3. In Section 4 we present experimental evidence that the rendering sequences generated by our algorithm indeed perform well, and conclude in Section 5.

2 Generating Universal Rendering Sequences

In this section we present two algorithms to generate universal rendering sequences. The first is inspired by classical space-filling curve constructions, and the second is obtained as a solution to an optimization problem.

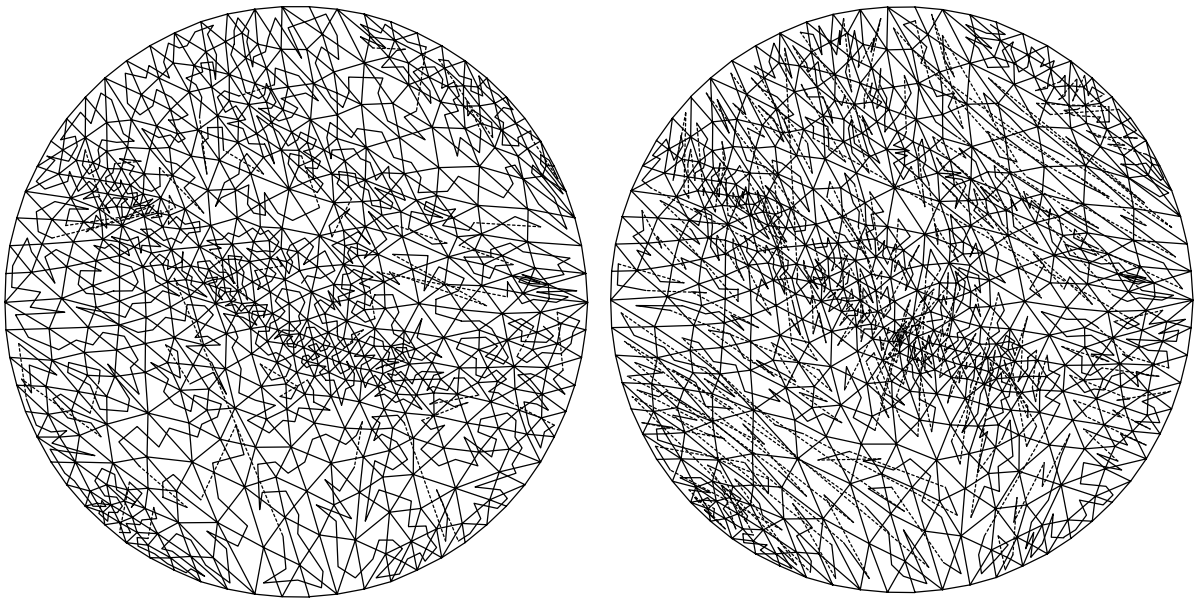
2.1 The Recursive Cut Algorithm

We know that the classical space-filling curves on rectangular grids (of sizes which are powers of two) have good locality properties, so as a first experiment it would be interesting to see how these curves perform as rendering sequences. It is easy to use the classical Hilbert curve to produce a rendering sequence for the regular triangle grid (instead of a rectangular grid). See Fig. 2b. The graph in Fig 4a shows the ACMR incurred by this rendering sequence as a function of the cache size. The Hilbert construction proceeds as follows: Partition the mesh into four (identical) quarters. Render all triangles in the first quarter (recursively), then all triangles in the second quarter (recursively), etc. The recursion terminates when the mesh contains only a few triangles. Care is exercised so that the last triangle rendered in the first quarter is adjacent to the first triangle rendered in the second quarter, and similarly for the third and fourth quarters. This is possible due to the regular structure of the mesh, and thus guarantees a continuous curve. The Hilbert curve construction inspires the following analogous recursive procedure for irregular triangle meshes: Partition the mesh into two approximately equal submeshes. Render the first submesh, then render the second submesh. Make sure, though, that the exit point from the first submesh is close to the entry point into the second submesh. To be more precise, we need to find a *balanced edge-cut* of the mesh connectivity graph, i.e. a set of edges of the graph such that removing them from the mesh results in two disconnected sets of vertices of approximately the same size. It is even better if the edge-cut contains only a small number of edges, because most of the cache misses will ultimately be incurred along the edge-cut. The procedure proceeds to first render the left submesh, then the triangles straddling the edge-cut, and then the right submesh.



(a) (b) (c) (d)

Figure 2: Possible rendering sequences for a regular triangle mesh generated from a 16x8 regular square grid. Note that all vertices but the boundary ones have degree 6. Dashed lines denote "jumps" in the sequence between non-neighboring triangles. (a) Simple "snake" raster. (b) Hilbert space-filling curve. (c) Sequence generated by the recursive cut algorithm. (d) Sequence generated by the MLA algorithm.



(a) (b)

Figure 3: Possible rendering sequences for an irregular mesh of 509 vertices and 950 triangles. Dashed lines denote "jumps" in the sequence between non-neighboring triangles. (a) Sequence generated by the recursive cut algorithm. (b) Sequence generated by the MLA algorithm.

Finding a balanced edge-cut of a graph is a much-studied problem in itself, with applications in parallel processing, numerical computation and VLSI, to name a few. Rather than reinvent the wheel, we used the excellent MeTis software package [13]. MeTiS is able to find balanced edge-cuts in time linear in the mesh size. Fig. 2c and 3a show the rendering sequences generated using this algorithm on two meshes, and Fig 4 the associated performance graphs. The first mesh is the regular triangulated grid mentioned before, where each vertex has degree 6.

The second mesh is irregular, whose vertices have degrees anywhere between 4 and 8. Since the rendering sequences depend on mesh connectivity alone, we visualize the irregular mesh using the graph drawing procedure of Tutte [23], where the boundary vertices are mapped to a circle, and each of the interior vertices is placed at the centroid of its neighbors (this drawing is generated by iteratively solving a system of linear equations for the planar coordinates of the vertices). To compare, Fig. 4 shows the performance curves generated by this algorithm for the meshes. For the regular mesh, the rendering sequence generated by the recursive cut algorithm is not much worse than the Hilbert curve. It is also possible to use the simple "raster snake" curve, which also seems to give reasonable results. Not surprisingly, a completely random rendering sequence performs dismally.

2.2 The Minimum Linear Arrangement Approach

Space-filling curves in general, and rendering sequences in particular, are attempts to impose a one dimensional ordering on a set of higher-dimensional elements. Indeed, space-filling curves are used for reducing higher-dimensional problems to one-dimensional ones without losing too much of the spatial correlations present in the data, e.g. in image compression [14].

Following this argument, the problem of generating an efficient rendering sequence may be cast as an instance of the *Minimum Linear Arrangement* (MLA) problem on hypergraphs. A *hypergraph* is a pair <V,HE> where V is a vertex set, and HE a set of *hyperedges* connecting sets of vertices. A graph is a special case of a hypergraph where every hyperedge connects just two vertices. The MLA problem requires that the n vertices of the hypergraph be mapped to the integers $\{1,..,n\}$, such that the sum of the hyperedge *lengths* is minimal. The length of a hyperedge $e=[v_1,..,v_k]$ is defined to be

$$L(e) = \max(m(v_1),..,m(v_k)) - \min(m(v_1),..,m(v_k)),$$

where $m:V\rightarrow\{1,..,n\}$ is the mapping function. This means that all vertices participating in a hyperedge should be mapped in close proximity. The MLA is a member of the class of *geometric embedding* problems [10], where combinatorial structures, e.g. graphs, are

embedded in a geometric domain, such that they optimize some geometric measure, e.g. distance. Here the graph is embedded into a one-dimensional grid.

Our mesh rendering problem may be cast as an instance of the MLA as follows: The hypergraph vertices correspond to the mesh triangles, and the hyperedges correspond to the mesh vertices, i.e. a hyperedge relates all mesh triangles incident on the same mesh vertex. The meaning of edge length in our context is the distance in the rendering sequence between the first and last triangles incident on the mesh vertex.

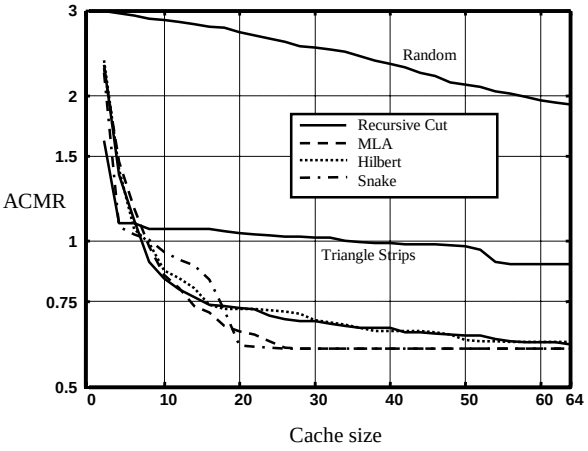
The MLA problem is known to be NP-Hard, hence efficient algorithms have been proposed to approximate the minimum. We use that of Bar-Yehuda et al. [3], which approximates the minimum in $O(n^{2.2})$ time and $O(n)$ space, where n is the number of vertices. The time complexity may be reduced by adjusting some algorithmic parameters, at the expense of the output quality.

Fig. 2d and 3b show the rendering sequences generated by the MLA algorithm on our two test meshes, and Fig. 4 compares the sequence performance with those generated by other methods. In particular, it is interesting to compare to the performance of a rendering sequence obtained from a leading triangle stripper [24]. While the ACMR of that sequence is reasonable, it is nowhere near optimal, since it was designed specifically for a cache of size two.

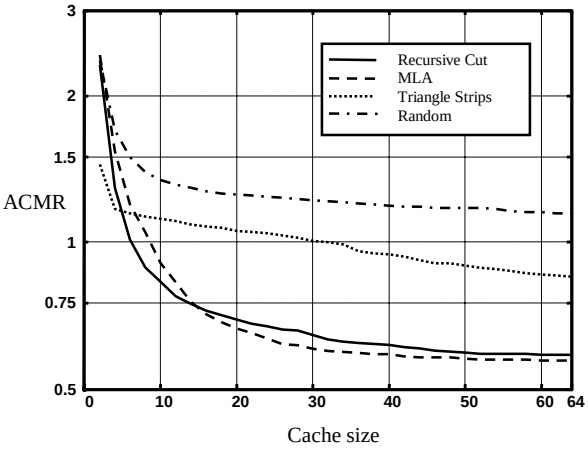
3 Application to Progressive Meshes

Many real-time 3D graphics applications, especially those dealing with large scenes, employ variants of the progressive mesh technique (also known as continuous level-of-detail) [11] to increase rendering performance. In a nutshell, this means that the polygon count of the scene is adjusted on the fly to adapt the geometric scene resolution to the rendered image resolution. For example, it is wasteful to render hundreds of polygons when the contribution of those polygons to the rendered image is less than one pixel.

Typical progressive mesh algorithms operate in two stages: In a preprocessing stage, a data structure is built containing information pertaining to the operations performed in order to increase or decrease the polygon count. The polygon count may be changed by a variety of methods, such as edge collapses [11] or vertex removals [19]. We consider the more general vertex removal method. The decision as to which vertex to remove at any given resolution level is usually based on geometric criteria, such as geometric approximation error relative to the original model. In essence, this means that at each level the vertex which least damages



(a)



(b)

Figure 4: Performance of different rendering sequences as a function of cache size. The random rendering sequence is the ordering of the triangles as they happened to appear in the VRML IndexedFaceSet shape in the input file. (a) Regular triangle grid (as in Fig. 2). Note how the snake raster is no worse, and sometimes even better, than the other sequences on very small or very large cache sizes. (b) Irregular triangle grid (as in Fig. 3).

the geometric shape of the model is removed first. The resulting data structure usually consists of a sequence of *update records* which record at each resolution which vertex is to be removed, and how the resulting hole is to be retriangulated. When increasing resolution, the same record indicates the vertex to be inserted to the mesh, and how the mesh connectivity is adjusted accordingly. During rendering, this information is used on the fly to adjust the mesh resolution according to some user (and view) dependent criteria.

Since the update records of the progressive mesh are generated offline, long before the rendering, this is a given for the rendering sequence generator. The mesh cannot be modified on the fly to optimize other criteria, such as the performance of the rendering sequence. Hence, given this sequence of update records, we must generate not only a rendering sequence for the highest resolution, but also a sequence of corresponding updates to the rendering sequence, so that it continues to perform well also at lower resolutions.

We experimented with two rendering sequence update methods. Assume a vertex is to be removed from the mesh, and that vertex is incident on triangles whose positions in the rendering sequence are $k_1,..,k_n$. In a manifold closed mesh, the resulting hole will have n-2 edges, hence the vertex removal will eliminate n triangles from the mesh, and retriangulating the hole will generate n-2 new triangles. The simplest way to update the rendering sequence is to arbitrarily assign the n-2 new triangles to the first n-2 indices of the n now available. This makes sense, as the new n-2 triangles fill in the same area as the old n triangles filled in the past, so the locality will be somewhat preserved. We call this the *simple* update algorithm. However, this will probably be suboptimal, and the assignment of these new triangles among the freed indices can be optimized to better preserve the locality. Towards this end, we developed the *smart* update algorithm, whose pseudo-code appears in Fig. 5. Note that we do not attempt more global updates to the rendering sequence, and the places of all mesh triangles not affected by the vertex removal are not changed in the rendering sequence. Fig. 6 shows a model at three levels of resolution, the rendering sequence for the highest resolution, and the rendering sequences generated for the two lower resolutions by the smart update algorithm.

It is obvious that the rendering sequence, updated as the resolution is decreased, will accumulate distortions such that after many updates it might cease to perform well. The key to the effectiveness of the procedure is graceful degradation of the performance. A good way to quantify the degradation is to compare the ACMR of the updated rendering sequence to that of a rendering sequence generated (using one of our methods) specifically for the given resolution. Fig. 7 plots the ACMR of the irregular mesh of Fig. 3 as a function of the triangle count, for a cache of size 16, and compares it to the ACMR that would have resulted had the rendering sequence been computed independently at each resolution. As is to be expected, the simple update algorithm performs quite poorly, accumulating significant distortion by the time a large number of vertices are removed. In contrast, the smart update algorithm performs almost as well as rendering sequences generated specifically for the mesh at that resolution, degrading very gracefully.

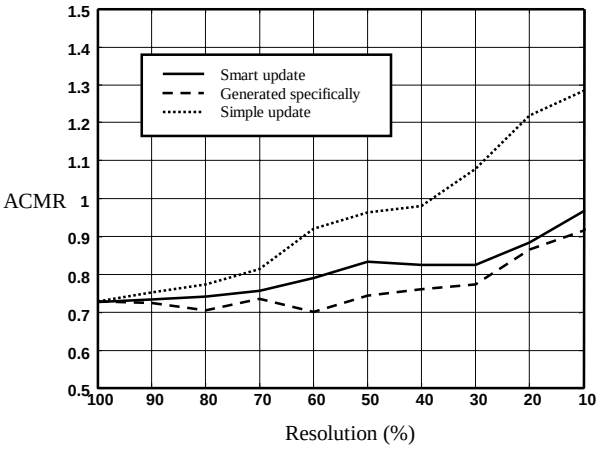


Figure 7: Performance of rendering sequence of irregular mesh (of Fig. 3) during geometric resolution reduction, comparing our update algorithms to what would have been obtained had the rendering sequence been generated specifically for each resolution. Vertex cache size = 16. The mesh simplification history, i.e. the sequence of vertex removals and retriangulations, was generated by a commercial simplifier available from www.virtue3d.com.

4 Experimental Results

We claim that both the methods presented above for generating rendering sequences are *universal* in the sense that they perform well for all cache sizes, and degrade gracefully when applied to progressive meshes with given simplification histories. To quantify this, we have run our algorithms on a variety of test meshes, and measured the ACMR empirically for different values of cache size and mesh resolution. Some of the vital statistics and algorithm runtimes on a 550MHz Pentium III PC with 128MB RAM for our test meshes appear in Table 1.

Fig. 8 shows plots of the ACMR of the test meshes, as a function of both cache size and mesh resolution, for the two rendering sequence generators described in Section 3. Quite surprisingly, the recursive cut algorithm generates rendering sequences which perform almost identically for all meshes, both as a function of cache size and as a function of mesh resolution. The MLA algorithm generates rendering sequences which exhibit some variance in the ACMR between meshes.

Figs. 8a and 8c show also the ACMR measured by Hoppe [12] for his algorithm run on the bunny4000 model (the famous Stanford bunny decimated to 4000

```
L – List of adjacency lists
L(t) - List of triangles that have com-
mon vertex with triangle t
S - Rendering sequence
S(t) - Index of triangle t in S
t1..tn – Triangles removed from mesh
T1..Tn-2 – Triangles inserted into mesh
S' – Updated rendering sequence

Input: S, t1..tn, T1..Tn-2
Output: S'

// Initialize S'
S' = S;
for i = 1 to n
  S'(ti) = empty;
endfor

// Assignment due to two common vertices
for i = 1 to n-2
  for j = 1 to n
    if Ti and tj have exactly 2 common
    vertices then
      S'(Ti) = S(tj);
      S(tj) = empty;
    endif
  endfor
endfor

// Initialization of adjacency lists
for i = 1 to n-2
  if S'(Ti) is empty
    for j = 1 to n
      if S(tj) is not empty and Ti and tj
      have at least one common vertex
        add tj to L(Ti);
      endif
    endfor
  endif
endfor
```

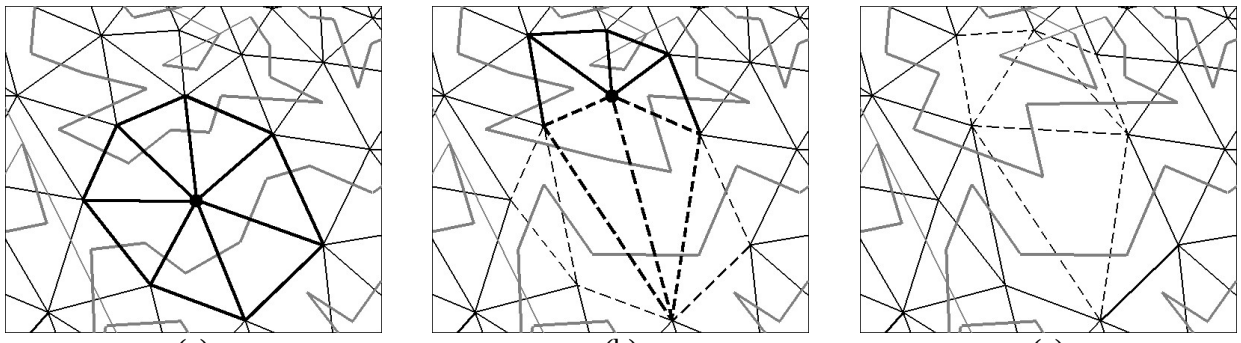
```
// Assignment due to one common vertex

while there exists at least one non-empty list in L
  let L(T) be a list with minimum length;
  let t = head of L(T);
  S'(T) = S(t);
  S(t) = empty;
  empty L(T);
  remove all occurences of t from other lists in L;
endwhile

// Assign the remaining faces to whatever places are
left

for i = 1 to n-2
  if S'(Ti) is empty
    for j = 1 to n
      if S(tj) is not empty
        S'(Ti) = S(tj);
        S(tj) = empty;
      endif
    endfor
  endif
endfor
```

Figure 5: Pseudo-code of the **smart** sequence update algorithm to account for a vertex removal.



(a) (b) (c)

Figure 6: Smart update of a rendering sequence during geometric resolution reduction. Thick lines mark the edges of the "star" whose center – the fat vertex - is to be removed. Dashed lines mark the edges of the retriangulated "hole". (a) High resolution. (b) One vertex removed. (c) Two vertices removed.

vertices). The values are slightly better than those of our rendering sequences. Recall, however, that Hoppe generates a *different* sequence for every cache size. Considering that we use just *one* universal sequence, our results are very competitive.

Table 2 shows the actual frame/sec rendering rates measured on a ASUS GeForce 2 graphics card containing a vertex buffer with ten entries, using our rendering sequence. This is compared to the frames rates achieved when an arbitrary rendering sequence is used, and the frame rates achieved by triangle stripping. It is evident that the frame rate speedup is quite consistent with that predicted by the ratio between the corresponding ACMR's, which can be up to factor 3.

5 Summary and Conclusion

This work describes two (related) methods to generate rendering sequences for meshes which should be very useful in the age of 3D hardware with vertex buffers. They may be precomputed once per mesh and then used for any cache size. These rendering sequences have also been shown to be useful in progressive mesh applications and apply to all types of 3D meshes, including non-manifold and non genus-0 meshes.

Software executables demonstrating the concepts described in this paper may be found at http://www.cs.technion.ac.il/~gotsman/caching.
An interesting result is that the rendering sequences seem to perform equally well on meshes of all sizes (as a function of cache size). This seems to be another positive indication of the "universality" of the sequences.

Future work will include improvements of the algorithms, optimization of the implementations, and more sophisticated updates to the progressive rendering sequence.

Acknowledgements

Thanks to Reuven Bar-Yehuda, Guy Even, Jon Feldman and Seffi Naor for helpful inputs on mesh partitioning and embedding algorithms, and making available their software for the MLA. The bunny and buddha meshes are courtesy of the Stanford University Computer Graphics Laboratory.

This work was partially supported by Israel-German fund (GIF) grant I-627-45.6/99 and the Technion VP for Research Fund.

Mesh	**Number of vertices**	**Number of triangles**	**Recursive cut runtime (sec)**
Regular32	561	1,024	0.3
Bunny (simplified)	1,092	2,084	0.6
Flipper	6,179	12,337	5.4
Face	12,530	24,118	9.5
Horse	19,851	39,698	18
Buddha	32,316	67,240	27
Bunny	34,834	69,451	32

Table 1: Characteristics of meshes used in our tests.

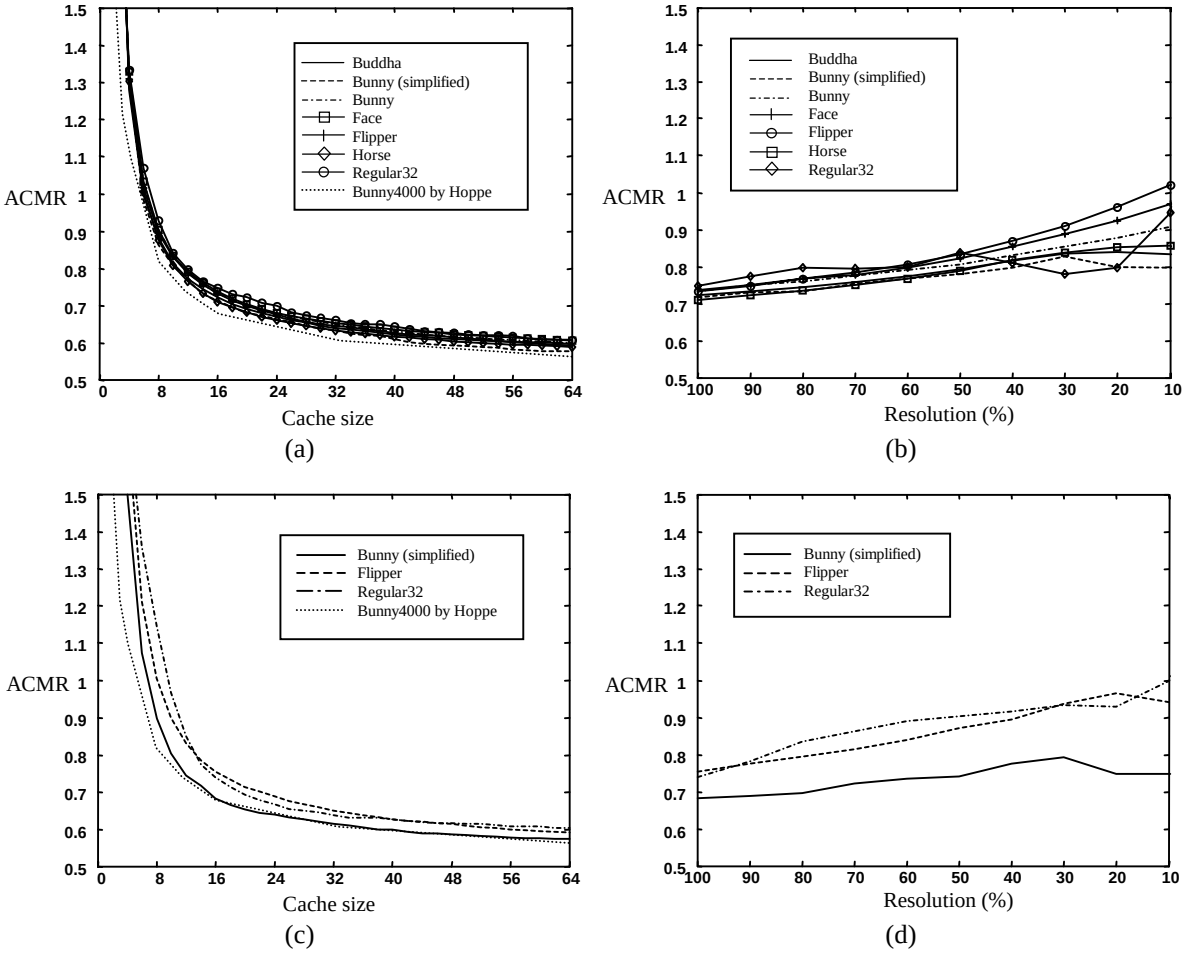


Figure 8: ACMR results on test meshes of Table 1. (a)-(b) Recursive cut algorithm. (c)-(d) MLA algorithm. The cache size in (b) and (d) is 16.

Mesh	ACMR			Rendering Speed (frames/sec)		
	Original	Rec. Cut	Triangle Strips	Original	Rec. Cut	Triangle Strips
Buddha	1.00	0.81	1.08	227	240	236
Bunny	1.00	0.83	1.08	228	229	185
Bunny simp.	2.76	0.81	1.06	1299	2439	2222
Face	1.34	0.83	1.10	418	680	546
Flipper	1.84	0.83	1.10	508	1299	787
Horse	2.83	0.81	1.09	98	327	205

Table 2. Rendering speedups using vertex buffer. The ACMR was measured by simulation of a vertex cache of size 10. The rendering speed was measured on the ASUS GeForce 2 GTS, which has an effective vertex cache size of 10 entries. “Original” – arbitrary rendering sequence (as appeared in original VRML file). “Rec Cut” – rendering sequence as generated by our recursive cut algorithm. “Triangle Strips” – rendering sequence generated by the triangle stripper of [24], and rendered using the OpenGL tri-strip primitive.

References

[1] Akeley, K. Haeberli, P., and Burns, D. The tomesh.c program. Available on SGI computers and developers toolbox CD. (1990).

[2] Bartholdi, J. J. and Goldsman, P., A continuous spatial index of a triangulated surface. *Ph.D. Thesis*, Industrial Engineering Dept., Georgia Inst. of Technology, 1999.

[3] Bar-Yehuda, R., Even, G., Feldman, J. and Naor, J. Computing an optimal orientation of a balanced decomposition tree for linear arrangement problems. Submitted, 2000. Software available at http://www.eng.tau.ac.il/~guy/Minla.

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

[5] Chow, M., Optimized geometry compression for real-time rendering. In *Visualization '97 Proceedings* (1997), IEEE, pp. 347-354.

[6] Deering, M., Geometry compression. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 13-20.

[7] El-Sana, J., Azanli, E., Varshney, A., Skip Strips: Maintaining triangle strips for view-dependent rendering. In *Visualization '99 Proceedings* (1999), IEEE.

[8] Evans, F., Skiena, S., and Varshney, A., Optimizing triangle strips for fast rendering. In *Visualization '96 Proceedings* (1996), IEEE, pp. 319-326.

[9] Gotsman, C., and Lindenbaum, M., On the metric properties of discrete space-filling curves. *IEEE Transactions on Image Processing*, Vol. 5, No. 5, pp. 794-797, 1996.

[10] Hansen, M. D., Approximation algorithms for geometric embeddings in the plane with applications to parallel processing problems. In *Proceedings of the Conference on Foundations of Computer Science*, (1989), IEEE, pp. 604-609.

[11] Hoppe, H., Progressive meshes. *Computer Graphics (SIGGRAPH '96 Proceedings)*, (1996), pp. 99-108.

[12] Hoppe, H., Optimization of mesh locality for transparent vertex caching. *Computer Graphics (SIGGRAPH '99 Proceedings)* (1999), pp. 269-276.

[13] Karypis, G., and Kumar, V., METIS – a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Ver. 4, University of Minnesota. Available on WWW at http://www-users.cs.umn.edu/~karypis/metis

[14] Lempel, A. and Ziv, J. Compression of two-dimensional data. *IEEE Trans. on Information Theory*, Vol 32, No. 1, pp.2-8, 1986.

[15] Lin, G. and Yu, T.P.-Y., A Non-recursive algorithm for minimum-time rendering of meshes with arbitrary genus. Preprint, RPI, 2001.

[16] Mitra, T. and Chiueh, T., A breadth-first approach to efficient mesh traversal. In *Workshop on Graphics Hardware Proceedings*, (1998), ACM.

[17] Orni, A. Measuring the locality of space-filling curves. *M.Sc. Thesis*, Computer Science Dept., Technion – Israel Inst. of Technology, 1998.

[18] Sagan, H. Space-filling curves. Springer Verlag, New York, (1994).

[19] Schroeder, W.J., Zarge, J. A. and Lorensen, W.E. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proceedings)*, (1992), pp. 65-70.

[20] Sowizral, H., Rushforth K. and Deering, M. The Java 3D API Specification (2nd Ed.). Sun Microsystems Press (Java Series), 2000.

[21] Stewart, J. Triangle strips for continuous level-of-detail meshes. *Graphics Interface 2001 Proceedings.*

[22] Sun Microsystems Elite3D Series. http://www.sun.com/desktop/products/Graphics/elite3djtf.html

[23] Tutte, W.T. How to draw a graph. *Proc. London Math Society*, Vol. 10 (1960), pp. 304-320.

[24] Xiang, X., Held, M., and Mitchell, J., Fast and effective stripification of polygonal surface models. In *Symposium on Interactive 3D Graphics Proceedings* (1999), ACM, pp. 71-78.