An improved vertex caching scheme for 3D mesh rendering

Article *in* IEEE Transactions on Visualization and Computer Graphics · August 2006

DOI: 10.1109/TVCG.2006.59 · Source: PubMed

2 authors:

Gang Lin
Verizon
34 PUBLICATIONS **2,564** CITATIONS
SEE PROFILE

Thomas Yu
Drexel University
39 PUBLICATIONS **492** CITATIONS
SEE PROFILE

An Improved Vertex Caching Scheme for 3D Mesh Rendering

Gang Lin and Thomas P.-Y. Yu

Abstract—Modern graphics cards are equipped with a vertex cache to reduce the amount of data needing to be transmitted to the graphics pipeline during rendering. To make effective use of the cache and facilitate rendering, it is key to represent a mesh in a manner that maximizes the cache hit rate. In this work, we propose a simple yet effective algorithm for generating a sequence for efficient rendering of 3D polygonal meshes based on greedy optimization. The algorithm outperforms the current state-of-the-art algorithms in terms of rendering efficiency of the resultant sequence. We also adapt it for the rendering of progressive meshes. For any simplified version of the original mesh, the rendering sequence is generated by adaptively updating the reordered sequence at full resolution. The resultant rendering sequence is cheap to compute and has reasonably good rendering performance, which is desirable to many complex rendering environments involving continuous rendering of meshes at various level of details. The experimental results on a collection of 3D meshes are provided.

Index Terms—Polygon mesh, rendering sequence, vertex cache, progressive mesh, efficient rendering.

1 Introduction

With the advent of 3D digitizers such as laser scanners, complex 3D meshes are increasingly available. These meshes can model complicated shapes in applications ranging from CAD/CAM design to video games to digital sculpture technologies [1] to bioinformatics; the problem of efficient rendering has been receiving the attention of many graphics scientists and practitioners. Modern graphics cards employ a built-in vertex buffer (or cache) to reduce the cost of transmitting the vertex data (including 3D coordinates, and optional color, normal, and texture coordinates) down to the graphics pipeline during rendering. Such an approach was first proposed by Deering [2] and further developed by Chow [3]. To make effective use of the vertex buffer and to speed up the rendering, the input mesh needs to be represented in a format that facilitates the vertex caching so as to maximize reusing the vertex data. Usually, the 3D meshes in various formats like VRML we encountered in practice do not have this property. Their face lists are arranged arbitrarily or in whatever order given by a 3D digitizer. Therefore, it is important to reorder the face list and generate a rendering sequence that minimizes the average cache miss ratio (ACMR), which is measured by the average number of cache misses incurred per face. In the following, we use the terms "vertex buffer" and "vertex cache" interchangeably.

A simple but important idea in efficient rendering is the observation that a long triangle strip can be rendered three times more efficiently than a straightforward triangle-by-triangle-based rendering. (Here, efficiency is measured in terms of the amount of geometry information needed to travel down the graphics pipeline.) This idea has been adopted in graphics API's such as OpenGL [4] and also in graphics processing hardware. When dealing with general triangle meshes, one must first "stripify" the mesh before the triangle strip idea can be applied. Under a certain formulation, the problem of finding an optimal stripification for a given mesh is an NP-complete problem [5], [6] and, currently, there are no known approximation algorithms. However, several heuristics-based fast algorithms have been constructed and reported to work well in practice [2], [3], [7], [8], [9], [10].

In particular, Hoppe [8] presented a greedy strip-growing and local optimization algorithm to construct good rendering sequence for a fixed-size FIFO cache. The method has been implemented in the D3DX library of Microsoft DirectX. Another closely related work by Bogomjakov and Gotsman [11] proposed a so-called universal rendering sequence for FIFO cache, which produces rendering sequences of higher ACMR compared to [8], but generalizes better to vertex cache with various sizes.

In this work, we propose a new algorithm for generating rendering sequence for arbitrary 3D polygonal meshes in both FIFO and controllable cache models using a greedy optimization approach. Our sequence yields better ACMR compared to other existing methods. We also extend the same idea to render a progressive mesh. For any simplified versions of the original mesh, we update the reordered rendering sequence (at full resolution) adaptively using the commands encoded in the progressive mesh. The resulting sequence has reasonably good performance and is very fast to compute. This is very useful to any application that requires rendering a mesh at fast and continuously changing resolutions.

2 Rendering Sequence Generation at Full Resolution

2.1 Methods

In our approach to generating a rendering sequence that minimizes ACMR, the problem was treated as a functional optimization and solved in a greedy fashion. Throughout

- *G. Lin is with the Department of Electrical, Computer, and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY 12180-3590. E-mail: ling@ecse.rpi.edu.*
- *T.P.-Y. Yu is with the Department of Mathematical Sciences, Rensselaer Polytechnic Institute, Troy, NY 12180-3590. E-mail: yut@rpi.edu.*

Manuscript received 1 Aug. 2005; revised 17 Dec. 2005; accepted 30 Dec. 2005; published 10 May 2006.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-0098-0805.

the paper, we consider the following cache models: 1) FIFO: first-in-first-out and 2) controllable: One can choose the vertex to enter the cache, as well as the vertex to leave the cache. During the rendering sequence generation, we color-code any vertex of the input mesh in the following three colors:

1. **White**: The vertex is not in the buffer. Some or all of its connecting faces (faces with the vertex as one of its bounding vertices) have not yet been rendered. Therefore, it will need to be pushed into buffer at subsequent rendering stages.
2. **Gray**: The vertex is currently in the buffer.
3. **Black**: The vertex has no more connecting faces that have not been rendered. The vertex can be either in or out of the cache. However, once it has left the cache, it will not be needed or come back again during the subsequent rendering procedure.

At each iteration of the proposed algorithm, we select one gray or white (in the case of empty buffer) vertex as the focus, denoted v_{focus}. For each face connecting to v_{focus} that has not been rendered, we push its white bounding vertex(es) into buffer and render it. Once all the faces connecting to v_{focus} have been rendered, we turn it into black. The blackened vertex may stay in the buffer until the due time for it to leave, depending on the cache model used. For FIFO cache, the leaving time is when the vertex reaches the front-most position of the buffer; for controllable cache, the vertex can leave immediately after it turns black. Apparently, the "vertex push" operation is only called upon when a connecting face of the concerning vertex is being drawn. Since there are no unrendered faces that connect any black vertex, a black vertex will never be needed for the subsequent rendering and, thus, will never enter the buffer again after it leaves. The above iteration continues until all vertices have turned into black. At this point, all the faces have been rendered, and the entire rendering process terminates. Like other methods [8], [11], [12], [13], the basic underlying idea is to preserve the locality as much as possible during the rendering. We approach it by focusing on one vertex at a time aiming at reducing vertex push operations, i.e., cache misses. The idea is simple and has been adopted before by [14], [15], [16], [17]. The key issue is really to determine which vertex should become the focus at each stage of rendering, which accordingly determines the order of the final rendering sequence. In order to achieve a rendering sequence with optimal rendering efficiency, it is not straightforward how the focus should be selected at various stages of rendering. This is the main focus of the present work. Our strategy is to associate each vertex v with a cost value, denoted $C(v)$, which is tailored to reduce cache misses. Specifically, the vertex with the minimum cost would be picked as v_{focus} at each step. The value of $C(v)$ depends on the following three factors:

1. **Number of vertex pushes needed**. This is the number of vertex pushes required for rendering all unrendered faces connecting to v, given whatever the current state of the buffer, i.e., which vertices are in the buffer, and what are their statuses (color, neighbors, adjacent faces, etc.). Before any push operation, we need to check if the buffer is full. If so, we have to pop one vertex out of the buffer in order to make room for the newcomers. Depending on the buffer replacement mechanism, the vertex to be popped can be in the front of the buffer in the FIFO case, or in any desired position of the buffer in the controllable case. Note that the least number of required pushes is the number of v's white neighbors since it is possible some of the needed vertices get popped out of buffer during the process. Moreover, the buffer operations (push and pop) during this iteration will affect the subsequent rendering iterations. Depending on the current status of the buffer, a vertex with fewer white neighbors is likely to be a good candidate for v_{focus}. We denote the total number of pushes that will be needed from this point on until turning black as C_1.
2. **Number of faces rendered**. This cost factor, denoted as C_2, is related to the number of faces that will be rendered during v turning black. Intuitively, the priority should be given to the vertex that leads to more rendered faces. The more faces rendered, the lower cost v would have.
3. **The cache position when turning black**. As described previously, upon turning black, v in the buffer may have to stay there until the due time for it to leave. It is desirable that a blackened vertex leaves earlier, so as to make room for other newcoming vertices. Ideally, v leaves the buffer as soon as it turns black. However, it is not always achievable, in practice, due to the constraints of cache model. Our third cost factor for selecting v_{focus}, denoted C_3, takes this into account. This factor varies by different cache models: 1) FIFO: Since the front vertex of the buffer leaves first, we set C_3 to be the distance from the position of vertex at the time of turning black to the front of the buffer. To make the factor independent of the size of the buffer, we normalize it by the buffer size. The smaller the distance, the earlier it will leave buffer, thus the smaller the cost. 2) Controllable: Since we have full control of which vertex will leave, C_3 is irrelevant. Therefore, we always set $C_3 = 0$ for controllable cache.

The final cost metric of v is the combination of the above three factors and can be expressed as

$$C(v) = k_1 C_1(v) + k_2 C_2(v) + k_3 C_3(v),$$

where k_1, k_2, and k_3 are weighting coefficients. How to set these weights for any given mesh is not straightforward. However, the following is sort of our intuition. For all manifold surface meshes, the number of faces is roughly two times of the number of vertices. Therefore, if we want to treat the first two factors C_1 and C_2 equally, we should set $k_1 \approx 2 \times k_2$, which will bring them into the same scale since C_1 is vertex-based and C_2 is face-based. Since $C_3 = 0$ in controllable cache, k_3 can be any value. However, in the FIFO case, it is not obvious how k_3 should be related to k_1 and k_2; thus, we have to resort to the approach of learning from examples, which is described later in experimental results section. The detailed description of how C_1, C_2, and C_3 are calculated is given in the next section.

Let V_g denote the vertex set currently in buffer. When $V_g \neq \emptyset$, we select the vertex in V_g with the minimum cost as v_{focus}, i.e., $v_{\text{focus}} = \arg\min_{v \in V_g} C(v)$. In the case $V_g \neq \emptyset$, we

Procedure K-**Cache-Reorder** (V, F, M)
1. Initialization: $V_w \leftarrow V$, $V_g \leftarrow \varnothing$, $V_b \leftarrow \varnothing$, $F_{\text{output}} \leftarrow \varnothing$
2. while $|V_b| < |V|$ /* iterate until all vertices turn into black */
3. if $V_g = \varnothing$ /* buffer is empty */
4. $v_{\text{focus}} \leftarrow$ white vertex with the minimum degree
5. else
6. $v_{\text{focus}} \leftarrow$ minimum cost vertex in V_g
7. $F'(v_{\text{focus}}) \leftarrow \{ f \in F \setminus F_{\text{output}} : v_{\text{focus}}$ is a bounding vertex of $f \}$
8. for each $f \in F'(v_{\text{focus}})$
9. $V'(f) \leftarrow \{$ white bounding vertex(es) of $f \}$
10. for each $v \in V'(f)$
11. if $|V_g| = K$ /* buffer is full */
12. **pop_one**(M, V_g, f)
13. Push v: $V_w \leftarrow V_w \setminus \{v\}$, $V_g \leftarrow V_g \cup \{v\}$
14. Output any renderable faces (i.e., all three bounding vertices are in buffer) except f
15. Output f: $F_{\text{output}} \leftarrow F_{\text{output}} \cup \{f\}$
16. Turn v_{focus} black: $V_b \leftarrow V_b \cup \{v_{\text{focus}}\}$; if $V_g \neq \varnothing$, $V_g \leftarrow V_g \setminus \{v_{\text{focus}}\}$
17. Checking each $v \in V_g$ for the possible turning black and leaving the buffer

Fig. 1. The proposed procedure for generating a rendering sequence.

pick a vertex from all white vertices that has the smallest degree.

2.2 Algorithm

Input: a polygonal mesh with:
V: vertices of the mesh
F: faces of the mesh
M: cache model (FIFO, Controllable)
Output: the rendering sequence F_{output}

Let K denote the cache size, V_w and V_b denote white and black vertices, respectively, and $|\cdot|$ denote the cardinality of a set. The proposed procedure for generating a rendering sequence is shown in Fig. 1.

Tie breaking. For simplicity, we did not specify the details on how to break ties in Steps 4 and 6 of the algorithm. For Step 4, it is the white vertex with the minimum degree that shows up first in V of the input mesh. For Step 6, it is the minimum cost vertex in V_g that is closest to the front of the buffer. Fig. 2 shows the procedure for popping one vertex out of the buffer.

In K-*Cache-Reorder*, Step 3 happens in two situations: 1) the initial stage of the entire procedure and 2) the initial iteration of each component if there are multiple connected components in the mesh. In Step 6, we pick the vertex with the minimum cost from the current buffer as focus. The following is how the score of a vertex in buffer v is calculated: The calculation of C_2 is straightforward, i.e., the number of unrendered faces that connect to v. To calculate C_1 and C_3, we tentatively take v as the focus and simulate the process based on the current status of the buffer, i.e., render all the unrendered faces connecting v until v turns black. During the simulation process, the number of vertex push operations will be counted as C_1, and C_3 will be derived from the buffer position of v when v turns black.

Initially, all faces connecting to any vertex are arranged in clockwise order starting with the one that appears first in the input face sequence. Therefore, the unrendered faces of the focus in Step 7, $F'(v_{\text{focus}})$, will maintain this order, but possibly with some gaps since some connecting faces of v_{focus} may have already been rendered. Notice that at the end of each iteration (Step 17), it is necessary to check each $v \in V_g$ for the possibility of turning black, i.e., all its connecting faces have already been rendered. Also, we need to check the possibility of popping vertices out of the buffer that are no longer needed in subsequent procedure. The vertices must be at the front of buffer in the FIFO case and can be anywhere in controllable cache.

In a case in which some isolated vertices (i.e., no faces connecting to them) are present in the input mesh, they will be picked as v_{focus} one-by-one at the beginning of K-*Cache-Reorder*, due to the smaller cost. Apparently, in this case $F'(v_{\text{focus}}) = \emptyset$ (Step 7), Steps 8 through 15 will be skipped, and v_{focus} turns into black directly from white. This is the only case when a vertex skips entering the buffer and changes color from white to black directly.

Procedure **pop_one**(M, V_g, f)
1. if M = "FIFO"
2. $v \leftarrow$ first vertex of V_g
3. else if M = "Controllable"
4. $v \leftarrow$ vertex with the maximum cost in V_g, and not a bounding vertex of f
5. $V_g \leftarrow V_g \setminus \{v\}$, $V_w \leftarrow V_w \cup \{v\}$

Fig. 2. The procedure for popping one vertex out of the buffer.

2.3 Some Properties

The above algorithm always terminates: The size of V_b increases by 1 (Step 16) or more (Step 17) at each step, so the while-loop in Step 2 terminates after almost $|V|$ iterations. The algorithm always produces a rendering sequence for an input mesh, i.e., a reordered face list of F.

At any instance of an execution of the algorithm, we have the following facts:

1. Any face with a bounding vertex $v \in V_b$ has been rendered (by definition of black vertex).
2. All vertices are colored black upon termination of the algorithm and, hence, by item 1, all faces are rendered upon termination.
3. The fate of each vertex in the mesh: It starts from the color white, then turns gray, white, gray..., until it finally turns black. As described earlier, the only case that the optional changing color is skipped would be that of isolated vertices. The number of changes into gray of a vertex is the number of "push" operations occurred to the vertex.

We assume the input mesh contains a constant number, p, of connected components, and denote the number of vertices and faces by $|V|$ and $|F|$, respectively. With the implementation using Deque data structure, the proposed $K\text{-}Cache\text{-}Reorder$ has time complexity $O((p+K)\times|V|+K\times|F|)$ in FIFO cache and $O((p+K)\times|V|+|F|)$ in controllable cache. Note that a vertex may change color from gray back to white, but the number of iterations, i.e., while-loop in $K\text{-}Cache\text{-}Reorder$, is at most $|V|$, which is the same as the number of times of picking focus. We omit the detailed breakdown analysis here.

3 Adaptively Update Rendering Sequence for Progressive Mesh

Many real-time 3D environments, especially those dealing with large, complex scenes, employ variants of the progressive mesh (PM) technique to accelerate rendering procedure. The mesh resolution is reduced using some kind of simplification technique whenever rendering the full-resolution mesh is deemed unnecessary, such as when the view window is too small to justify full-resolution rendering. This is called level-of-detail (LOD) based rendering. There are basically two types of LOD schemes: 1) *discrete*, where a fixed number of levels of detail are involved, and 2) *continuous*, where the change of detail in the scene is required to be generated continuously on-the-fly based on the view and scene parameters [18]. Obviously, a simplified mesh in the hierarchy reduces the amount of geometric data needing to be transmitted to the rendering pipeline. Similar to the full resolution mesh, the rendering sequence at any other resolution levels can be reordered to achieve high cache coherence. Some prior works have been proposed to address this, such as [11], [19], [20]. In the following, we extend the proposed rendering sequence generator to the progressive mesh.

Many mesh simplification algorithms have been proposed in literature, which can be categorized into vertex-based [18], edge-based [19], [20], [21], and face-based [22]. Briefly, at each step, the algorithm chooses an element to be simplified that least damages the geometric shape of the mode depending on the defined error metrics or some characteristics, e.g., surface curvature. Consequently, a series of update records are generated that indicate which element is to be removed and how the affected region should be filled (retriangulated). The progressive mesh we used in this work is based on the vertex-removal method. At each step of simplification, one vertex is removed, n faces with the vertex as bounding vertex will be removed, and $m(m<n)$ new faces will be generated to fill the hole due to the removal. The update records of the progressive mesh are generated offline. Based on these records, the idea is to generate a rendering sequence for a simplified mesh at any resolution by updating the reordered sequence at full resolution by our $K\text{-}Cache\text{-}Reorder$, such that the resultant sequence for progressive meshes maintains good vertex cache coherence.

The following is our strategy: First, the command of "vertex removal" and "face removal" is carried out straightforwardly, i.e., the vertices are removed from the vertex list of the mesh, and the faces are removed from rendering sequence. The key is to insert the new faces into proper positions so that the resultant rendering sequence preserves the cache coherence. The simple way to update the rendering sequence is just to insert the new faces at the empty positions sequentially that have been made available due to face removal. Apparently, the method does not preserve well the locality, and thus degrades significantly the rendering efficiency. We propose a better update approach that takes advantage of the scalable properties of our rendering sequence, i.e., the locality of every single face in our sequence is known explicitly. Recall that each vertex turns into black at some point during $K\text{-}Cache\text{-}Reorder$; this enables us to insert new faces in such a manner: Each new face, f, is inserted to the position of the sequence where one of f's three bounding vertices was first picked as v_{focus} and turned black. Therefore, any new faces will be put into places in the sequence where some of their neighboring faces are located, thus preserving locality.

The update procedure is fast. By using linked-list data structure, the time complexity of generating sequence is $O(D_v+D_f+A_f)$, where D_v, D_f, and A_f denote the number of removed vertices, removed faces, and new faces. Apparently, it is much more efficient computationally than applying $K\text{-}Cache\text{-}Reorder$ to each simplified mesh independently.

4 Experimental Results and Discussions

The proposed algorithm was implemented using C++, and the executables can be downloaded.[1] We evaluate the rendering performance on a collection of data sets consisting of 30 popular 3D meshes in VRML text format. All tests were carried out using a Windows PC with an AMD Athlon 2.09 GHz CPU and 1.0 GB of RAM. Fig. 3 compares the ACMR of the proposed rendering sequence in both FIFO and controllable cache of size 12 with a well-known universal sequence by BoG [11] and Hoppe [8] method optimized for cache size 12. Our algorithm yields lower ACMR than both methods in FIFO cache. As expected, the additional flexibility of controllable cache achieves better ACMR. To our knowledge, there is no such controllable cache existing yet; it is not clear whether the gained rendering efficiency would justify the higher complexity of hardware realization it induces. It is also worth mentioning that ACMR variation among 30 test meshes indicated by standard deviation in Fig. 3 suggests that the

1. K-Cache-Reorder: http://www.cs.rpi.edu/~ling/K-Cache-Reorder.

3D Mesh	Vertex#	Face#	ACMR (12)					
			Random	Original	BoG	Hoppe	LY-FIFO	LY-Control
chain	240	480	2.883	0.575	0.721	0.575	0.588	0.500
dolphin	287	563	2.874	0.950	0.787	0.698	0.650	0.602
sphere	380	756	2.931	1.029	0.765	0.631	0.590	0.611
venus	711	1419	2.958	0.963	0.764	0.691	0.655	0.600
eight	766	1536	2.962	0.878	0.803	0.663	0.633	0.635
torus	800	1600	2.966	1.388	0.794	0.648	0.613	0.611
teapot	2022	3751	2.986	1.010	0.755	0.665	0.648	0.631
shape	2562	5120	2.987	0.906	0.794	0.617	0.589	0.639
beethoven	2655	5028	2.990	0.975	0.779	0.697	0.658	0.637
cow	3066	5804	2.993	1.007	0.794	0.674	0.652	0.635
cat	3644	7163	2.991	1.500	0.768	0.709	0.664	0.629
female-toso	4654	9095	2.992	1.787	0.783	0.657	0.640	0.629
distcap	6253	12339	2.995	1.535	0.770	0.688	0.650	0.625
fandisk	6475	12946	2.996	1.023	0.794	0.640	0.625	0.625
zahn-a	6586	13168	2.994	2.051	0.748	0.732	0.646	0.583
blob	8036	16068	2.998	1.072	0.784	0.662	0.640	0.617
mannequin	11704	23402	2.998	0.935	0.794	0.636	0.619	0.616
face	12530	24118	2.997	1.366	0.788	0.694	0.657	0.644
dinosaur	14070	28136	2.998	2.853	0.765	0.654	0.652	0.610
aircraft-f14	15111	29261	2.998	1.020	0.788	0.644	0.634	0.632
toilet	22926	45864	2.999	2.883	0.763	0.710	0.650	0.612
david	24085	47753	2.999	2.430	0.766	0.720	0.655	0.617
sculpture	25386	50780	2.999	2.941	0.766	0.701	0.649	0.609
foot	25845	51690	2.999	1.129	0.785	0.637	0.623	0.630
skull	33419	65244	2.999	1.828	0.781	0.661	0.642	0.643
bunny	35947	69451	2.999	2.085	0.789	0.641	0.628	0.635
horse	48485	96966	2.998	2.777	0.786	0.657	0.630	0.631
feline	49864	99732	2.954	1.376	0.772	0.692	0.649	0.621
dragon	437645	871414	2.747	1.725	0.762	0.708	0.653	0.616
buddha	543652	1087716	2.732	1.656	0.759	0.709	0.651	0.610
Average			**2.964**	**1.522**	**0.775**	**0.670**	**0.638**	**0.618**
Stddev			**0.069**	**0.683**	**0.017**	**0.035**	**0.021**	**0.026**

Fig. 3. Comparing the ACMR of five rendering sequences at a cache size of 12: Random (randomly permutated face sequence), Original (the face sequence given by the original mesh), BoG [11], Hoppe [8], and the proposed LY-FIFO and LY-Control.

Hoppe sequence has the greatest variation and the BoG sequence is most stable. Our FIFO sequence lies between the above two, and controllable cache sequence has slightly higher ACMR variation.

Fig. 4 shows a partially rendered Venus model, where 12 blue vertices are currently in cache. Fig. 5 illustrates the cache misses of four different rendering sequences using the visualization tool provided by [11]. The proposed algorithm is easy to implement, and runs fast. The running time on a set of test meshes with various sizes are shown in Fig. 6.

Fig. 4. Illustrating 20, 50, and 80 percent rendered Venus model by the proposed K-$Cache$-$Reorder$ using FIFO cache size 12. Any faces shown in green are already drawn, and others (gray and red) are unrendered faces. The blue dots indicate the vertices currently in cache.

The size of the built-in cache varies on graphics processing units (GPUs) of different models and vendors. For example, Nvidia GeForce FX 5600 has a cache of size 20, with an effective size of 16. We tested the performance of the proposed rendering sequence on different cache sizes. Fig. 7 shows the ACMR comparison in both FIFO and controllable cache and how they vary with the different cache sizes. As expected, the ACMR decreases consistently with the increases of the cache size in both cache models. However, the difference between FIFO and controllable sequences narrows with the increases of the cache size, which suggests that controllable cache yields greater improvement when cache size is small.

Furthermore, the curve in Fig. 7 demonstrates that the performance gain from the increase of cache size decreases rapidly beyond a certain buffer size, around 32. Therefore, we expect that the vertex cache will become bigger in the future with the advancing of GPU, but only to some degree since, after reaching a certain size, the induced improvement may not worth the added cost of hardware and the time to maintain the cache, as also suggested by [19].

As described previously, the cost optimization function in the proposed algorithm has three weighting coefficients k_1, k_2, and k_3 indicating the relative importance of three factors. We set them empirically as 1.0, 0.5, and 1.3, respectively, in our experiments. To demonstrate how their variation affects rendering performance, we generated the sequences using various combinations of two parameters

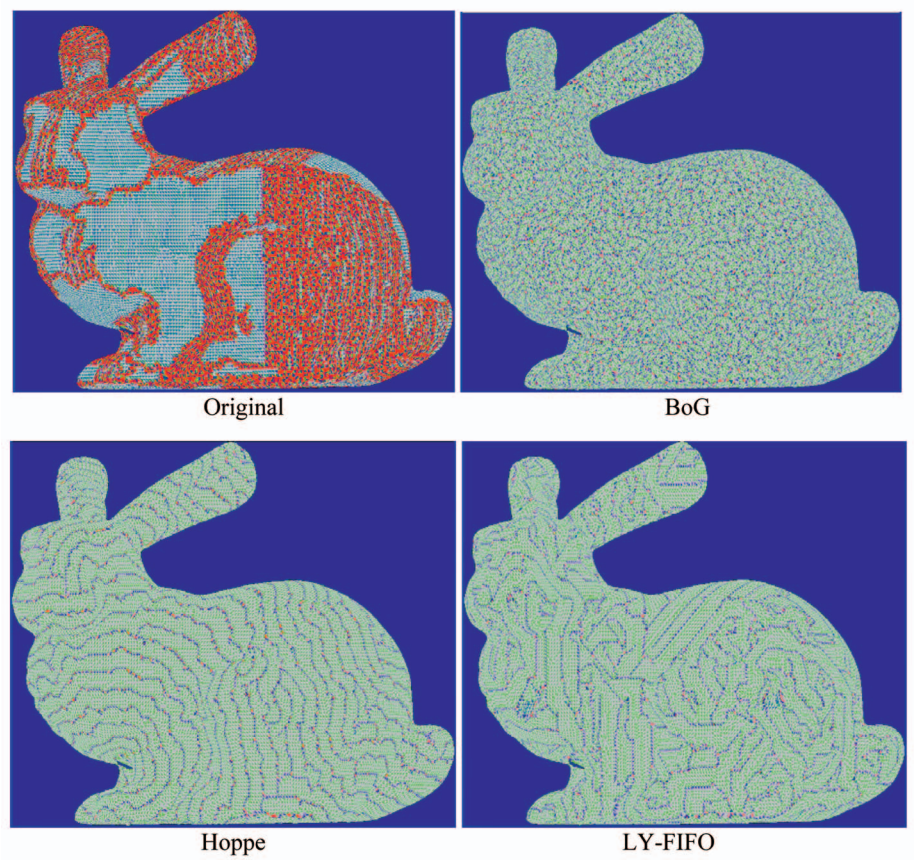

Fig. 5. The cache misses of Bunny using four rendering sequences in FIFO cache of size 12: original, BoG, Hoppe, and LY-FIFO. Their corresponding ACMR are 2.085, 0.789, 0.641, and 0.628, respectively. Gray indicates cache hits. Green indicates first cache miss. Purple indicates second cache miss. Red indicates the third or more cache miss. (Bogomjakov and Gotsman [11] provided the visualization tool.)

when $k_1 = 1.0$ is fixed. The surface plot in Fig. 8 shows the average ACMR on 30 test meshes at different k_2 and k_3.

For initial focus in K-$Cache$-$Reorder$, we pick the vertex with the minimum degree, i.e., number of its white neighbors. This approach has been adopted and shown to have good performance in solving the problem of reducing the bandwidth of sparse matrices [23] or the cut width of graph separators [24]. However, it is not clear how it impacts in the context of reducing cache misses in this work. To find out, we investigated experimentally how different initial focus affects the final rendering performance by randomly selecting 20 vertices from the mesh and generating the rendering sequence using them as initial focus. Fig. 9 gives some statistics of ACMR based on these sequences, which suggests that the initial focus selection has little effect on the rendering efficiency of our final sequence.

As described in [11], universality is a useful feature of a rendering sequence. It reflects the rendering performance of a sequence optimized for certain cache size when applied to various cache sizes. We generated rendering sequences on the set of meshes based on cache size 12 and simulated the rendering of these sequences on six different cache sizes.

3D Mesh	Vertex#	Face#	Running Time (second)							
			K=10	*K*=12	*K*=16	*K*=24	*K*=32	*K*=64	*K*=96	*K*=128
dolphin	287	563	0.02	0.02	0.03	0.02	0.03	0.02	0.03	0.03
venus	711	1419	0.03	0.03	0.03	0.05	0.06	0.11	0.13	0.16
teapot	2022	3751	0.08	0.06	0.09	0.13	0.16	0.22	0.24	0.25
shape	2562	5120	0.11	0.14	0.20	0.28	0.34	0.72	1.38	1.41
beethoven	2655	5028	0.09	0.11	0.16	0.19	0.25	0.42	0.58	0.73
cow	3066	5804	0.11	0.14	0.17	0.23	0.28	0.52	0.67	0.95
blob	8036	16068	0.34	0.44	0.55	0.70	0.92	1.77	2.80	3.69
mannequin	11704	23402	0.53	0.67	0.88	1.19	1.58	3.14	4.92	6.69
face	12530	24118	0.59	0.73	0.88	1.03	1.30	2.41	3.61	5.06
dinosaur	14070	28136	0.70	0.83	1.02	1.19	1.48	2.70	3.69	4.66
david	24085	47753	1.92	2.20	2.05	2.39	2.73	4.66	7.02	9.52
bunny	35947	69451	2.38	3.03	3.16	3.92	4.70	8.94	14.47	20.31
horse	48485	96966	3.23	4.14	4.28	5.20	6.42	11.34	18.05	24.02
dragon	437645	871414	181.83	194.06	179.83	149.69	133.69	140.42	166.53	205.44
buddha	543652	1087716	279.64	274.56	238.66	183.86	167.25	155.48	199.63	248.77

Fig. 6. The running times of our K-$Cache$-$Reorder$ in FIFO cache of various sizes.

Cache Size		K=10	K=12	K=16	K=24	K=32	K=64
Average ACMR	FIFO	0.664	0.638	0.606	0.579	0.566	0.543
	CONTROL	0.635	0.618	0.594	0.571	0.557	0.536

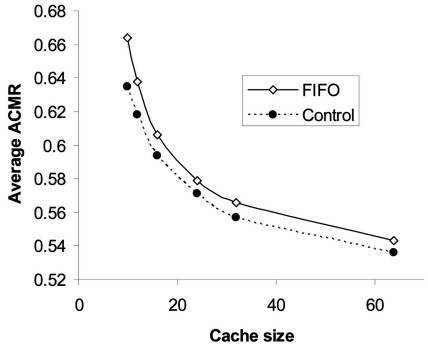

Fig. 7. The average ACMR on 30 test meshes using the proposed K-$Cache$-$Reorder$ based on FIFO and controllable cache of six different sizes.

Fig. 10 compares the universality of three methods, which shows BoG sequence has the best universality, and Hoppe's and the proposed sequence are close.

For the proposed adaptive sequence updating for progressive mesh, we demonstrate the rendering efficiency by comparing it with other two methods: 1) simple update, i.e., insert faces sequentially at the available empty spaces, and 2) generate a sequence specifically at each resolution using K-$Cache$-$Reorder$. Fig. 11 shows ACMR curves of three sequences for "Cow" and "Mannequin" at various resolutions. As can be seen, the simple update approach yields ACMR that increases rather drastically with the decreasing of mesh resolution. In contrast, the sequence by separately applying K-$Cache$-$Reorder$ has close to constant ACMR. Our adaptive update approach lies in between. Its running time on different resolutions of four meshes is shown in Fig. 12, which is significantly faster than specifically applying K-$Cache$-$Reorder$. Notice that the running time increases with resolution varying from 80, 50, to 20 percent. This is because our sequence updating at

3D Mesh	Vertex#	ACMR using 20 random initial vertices			
		Average	Min	Max	Stddev
Venus	711	0.65528	0.653277	0.655391	0.000485
Cow	3066	0.652028	0.65162	0.654721	0.000701
Mannequin	11704	0.619322	0.619178	0.621913	0.000627
Bunny	35947	0.628079	0.628054	0.628126	2.48E-05
Dragon	437645	0.653256	0.653158	0.654203	0.00023

Fig. 9. The ACMR of the proposed rendering sequence based on 20 randomly selected vertices as initial focus in a FIFO cache of size 12.

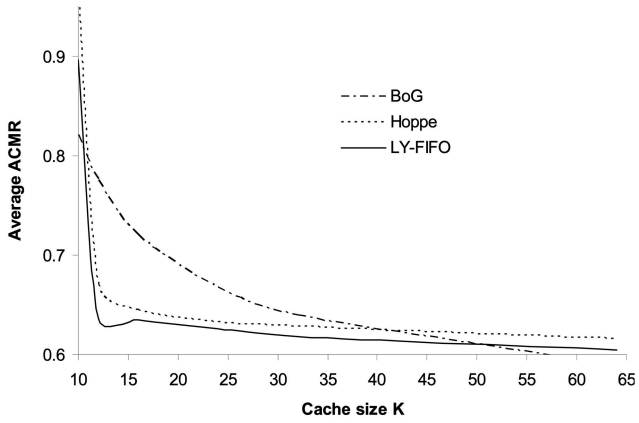

Fig. 10. The universality of three rendering sequences: BoG, Hoppe, and LY-FIFO. The sequences were all generated based on a fixed cache size 12, and then used to simulate rendering in six different cache

three resolutions all started from the original full resolution. This may not be necessary, in practice, since any mesh at current resolution can be directly updated from the one preceding to it in progressive meshes.

Comparing with specific K-$Cache$-$Reorder$ at each resolution, the adaptive update has lower computational complexity. Therefore, it trades rendering efficiency for the running speed, which is desirable for many time-critical applications that involve frequent changing rendering resolution on-the-fly. However, in many of the real-time complex LOD rendering environments such as view-dependent rendering, the system CPU is very often heavily loaded, such as

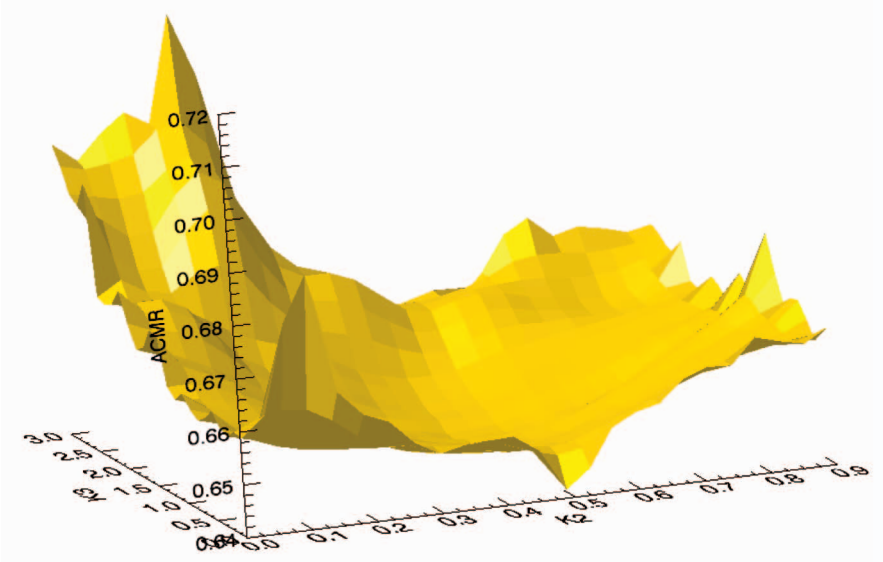

Fig. 8. Average ACMR of the proposed sequences on 30 test meshes using various parameters K_2 and K_3, when $K_1 = 1.0$ is fixed.

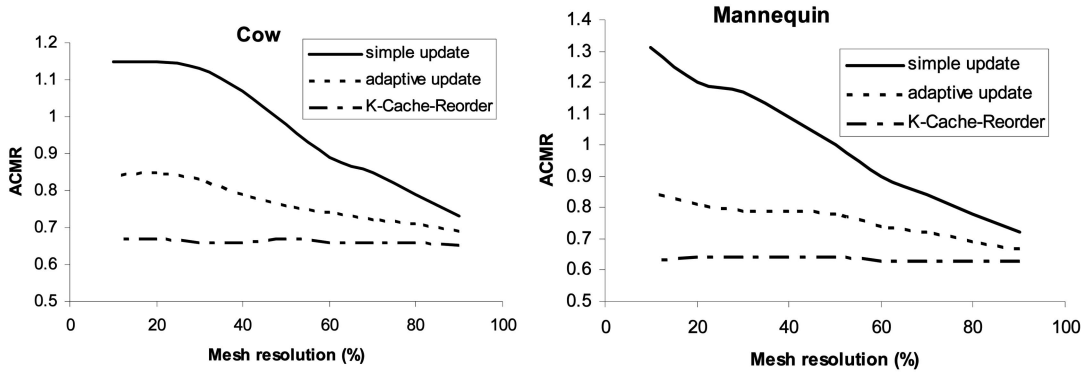

Fig. 11. Comparing ACMR of the rendering sequences for progressive mesh "Cow" (3,066 vertices) and "Mannequin" (11,704 vertices) in FIFO cache size of 12 by three methods: 1) simple update, 2) adaptive update, and 3) the sequence generated specifically by $K\text{-}Cache\text{-}Reorder$.

dynamically determining the display vertices based on view and illumination-dependent parameters, retriangulating the mesh, etc. Any sequence update methods including the proposed one introduce some additional computation to the system. Therefore, from a practical rendering perspective, it is necessary to determine the actual bottleneck of the rendering pipeline on the specific system and obtain the optimal trade-off between the amount of workload assigned to sequence updating and the rest of the pipeline, in order to achieve the overall gain of rendering speed. This type of testing and scheduling is beyond the scope of the present paper. It has been demonstrated [19] that the actual rendering is significantly faster by dynamically maintaining triangle strips using Skip Strips data structure. With the rapid development of today's video card, GPU has become increasingly powerful and programmable [21]. It has been used for general-purpose computation, and serves as a useful coprocessor in addition to the system CPU. One can imagine many of the CPU workloads can be potentially shifted to GPU in some CPU-limited rendering systems. Therefore, looking forward, we believe the extra cost on CPU that any dynamic sequence update for progressive meshes introduces, including the present method, will become negligible in the future.

This paper explores non-triangle-strip-based greedy algorithms, in both controllable and FIFO cache settings, for the problem of efficient rendering sequence generation of arbitrary 3D polygonal meshes. The first contribution of this paper is a simple and fast algorithm for rendering sequence generation based on functional optimization, which outperforms the current existing methods in terms of ACMR. The second contribution is that we derived an adaptive update scheme to generate a rendering sequence for progressive meshes based on our reordered sequence. This approach has low computational complexity, and achieves reasonably good ACMR at various resolutions.

Throughout the paper, we used ACMR measure to demonstrate the performance of the proposed rendering sequence. It is well known that there are several potential sources of bottleneck in GPU pipeline during the rendering process, such as geometry transferring, vertex shader, rasterizer, and fragment processing. It has been reported [11] that the actual rendering speed measured by frame rate correlates well with the ACMR, if the bottleneck of the rendering pipeline is indeed at vertex fetching and processing.

We address only the part of efficient rendering pertaining to geometry model, and we have not addressed issues in raster images (e.g., texture and bump maps) used in shading a geometric model. It will be interesting to explore how the techniques studied in this work interact with those pertaining to rasterization. Although the problem of efficient rendering of meshes with irregular connectivity are highly challenging, the connectivity information is not the most essential information of the geometric content of the mesh. Indeed, the geometric content in a mesh is unchanged if we perturb a vertex along the tangential

3D Mesh	Vertex#	Face#	Resolution	Deleted Vertices	Deleted Faces	Inserted Faces	Time (second)
Cow	3066	5804	80%	581	3475	2315	0.016
			50%	1467	8618	5716	0.047
			20%	2389	13487	8844	0.047
Mannequin	11704	2340	80%	2340	13644	8964	0.078
			50%	5851	33369	21667	0.14
			20%	9361	52306	33584	0.203
Bunny	35947	69451	80%	6957	41195	27304	0.266
			50%	17377	102464	67738	0.469
			20%	27811	162469	106908	0.687
Buddha	543652	1087716	80%	108772	598959	381415	4.484
			50%	271929	1505322	961464	7.688
			20%	435086	2401098	1530926	11.406

Fig. 12. The running time of adaptive sequence updating on progressive meshes at three resolutions: 80, 50, and 20 percent.

direction of the underlying surface. Several algorithms have been proposed for remeshing a mesh with irregular connectivity into one with subdivision connectivity. It might be useful to explore efficient rendering methods optimized for such kind of mesh representations.

Finding the optimal rendering sequence that minimizes vertex cache misses is a hard combinatorial optimization problem. This work presents a simple yet very effective approach to tackle it. Many questions, like whether or not some other factors can be exploited besides the ones we used in our cost metric and how to adjust the weighting coefficients systematically to optimize the rendering efficiency for any given specific graphics hardware realization, deserve further investigation.

ACKNOWLEDGMENTS

The authors would like to thank Alex Bogomjakov, Hugues Hoppe, John Gilbert, and Craig Gotsman for useful pointers. They also want to thank the anonymous reviewers for their valuable suggestions. They acknowledge the following sources from which they downloaded the meshes used in this work: http://graphics.stanford.edu/software/scanview/models, http://graphics.cs.uiuc.edu/~garland/CMU, http://www.cc.gatech.edu/projects/large_models, and http://www.cs.technion.ac.il/~gotsman.

REFERENCES

[1] M. Levoy, "The Digital Michelangelo Project," *Computer Graphics Forum,* vol. 18, no. 3, pp. 13-16, 1999.
[2] M. Deering, "Geometry Compression," *Proc. 22nd Ann. Conf. Computer Graphics and Interactive Techniques,* 1999.
[3] M.M. Chow, "Optimized Geometry Compression for Real-Time Rendering," *Proc. Eighth Conf. Visualization,* 1997.
[4] J. Neider et al., *OpenGL Programming Guide: The Official Guide to Learning OpenGL,* release 1. Addison-Wesley, 1993.
[5] M.B. Dillencourt, "Finding Hamiltonian Cycles in Delaunay Triangulations is NP-Complete," *Discrete Applied Math.,* vol. 64, no. 3, pp. 207-217, 1996.
[6] R. Estkowski, J.S. B. Mitchell, and X. Xiang, "Optimal Decomposition of Polygonal Models into Triangle Strips," *Proc. 18th Ann. Symp. Computational Geometry,* 2002.
[7] F. Evans, S. Skiena, and A. Varshney, "Optimizing Triangle Strips for Fast Rendering," *Proc. Seventh Conf. Visualization,* 1996.
[8] H. Hoppe, "Optimization of Mesh Locality for Transparent Vertex Caching," *Proc. 26th Ann. Conf. Computer Graphics and Interactive Techniques,* 1999.
[9] X. Xiang, M. Held, and J. Mitchell, "Fast and Effective Stripification of Polygon Surface Models," *Proc. Symp. Interative 3D Graphics,* 1999.
[10] P. Diaz-Gutierrez et al., "Constrained Strip Generation and Management for Efficient Interactive 3D Rendering," *Proc. Computer Graphics Int'l Conf. (CGI),* 2005.
[11] A. Bogomjakov and C. Gotsman, "Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes," *Computer Graphics Forum,* vol. 21, no. 2, pp. 137-148, 2002.
[12] Y. Zhu, "Uniform Remeshing with an Adaptive Domain: A New Scheme for View-Dependent Level-of-Detail Rendering of Meshes," *IEEE Trans. Visualization and Computer Graphics,* vol. 11, no. 3, pp. 306-316, May-June 2005.
[13] Z. Karni, A. Bogomjakov, and C. Gotsman, "Efficient Compression and Rendering of Multi-Resolution Meshes," *Proc. Conf. Visualization,* 2002.
[14] G. Lin and T.P. Yu, "Greedy Algorithms for Rendering Sequence Generation of Triangle Meshes for Controllable and FIFO Cache Models," manuscript, 2004.
[15] E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," *Proc. 1969 24th Nat'l ACM Conf.,* 1969.
[16] P. Alliez and M. Desbrun, "Valence-Driven Connectivity Encoding for 3D Meshes," *Proc. Eurographics Conf.,* 2001.
[17] S. Sloan, "An Algorithm for Profile and Wavefront Reduction of Sparse Matrices," *Int'l J. Numerical Methods Eng.,* vol. 23, pp. 1315-1324, 1986.
[18] W.J. Schroeder, J.A. Zarge, and W.E. Lorensen, "Decimation of Triangle Meshes," *Proc. 19th Ann. Conf. Computer Graphics and Interactive Techniques,* 1992.
[19] H. Hoppe, "Progressive Meshes," *Proc. 23rd Ann. Conf. Computer Graphics and Interactive Techniques,* 1996.
[20] M.-E. Algorri and F. Schmitt, "Mesh Simplification," *Proc. Eurographics Conf.,* 1996.
[21] R. Ronfard, J. Rossignac, and J. Rossignac, "Full-Range Approximation of Triangulated Polyhedra," *Proc. Eurographics Conf.,* 1996.
[22] A.D. Kalvin and R.H. Taylor, "Superfaces: Polygonal Mesh Simplification with Bounded Error," *IEEE Computer Graphics and Applications,* vol. 16, no. 3, pp. 64-77, May 1996.
[23] N. Gibbs, W.J. Poole, and P. Stockmeyer, "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix," *SIAM J. Numerical Analysis,* vol. 18, no. 2, pp. 235-251, 1976.
[24] J. Diaz, J. Petit, and M. Serna, "A Survey of Graph Layout Problems," *ACM Computing Surveys,* vol. 34, no. 3, pp. 313-356, 2002.

Gang Lin received the bachelor's degree in environmental engineering from the Anhui University of Science and Technology in 1993, the master's degree in engineering mechanics from the China University of Mining and Technology at Beijing in 1996, and the PhD degree in electrical engineering from the Beijing University of Posts and Telecommunications in July 1999. He was a postdoctoral research associate in the Mathematical Sciences and Computer Science Department at Rensselaer Polytechnic Institute (RPI) from October 1999 to April 2002. Since May 2002, he has been a research associate jointly with University of Arizona and the Electrical Computer and Systems Engineering department of RPI. His research interests include image and video compression, pattern recognition, machine learning, 3D computer graphics and visualization, and biomedical image analysis.

Thomas P.-Y. Yu received the BS degree in computer science and the MS degree in mathematics from Purdue University in 1993. He received the PhD in scientific computing and computational mathematics from Stanford University in 1997. From 1997-1998, he was a postdoctoral fellow in the Department of Statistics at Stanford University. He taught at the Department of Mathematical Sciences at Rensselaer Polytechnic Institute from 1998 to 2005 before he joined Drexel University. He works in the area of multiscale method for geometric modeling and nonlinear signal processing. In 2000, he was granted a US National Science Foundation Career Award for his work in this area.