Fast Triangle Reordering for Vertex Locality and Reduced Overdraw

Pedro V. Sander
Hong Kong University of Science and Technology

Diego Nehab
Princeton University

Joshua Barczak
3D Application Research Group, AMD

(a) Lin and Yu [2006] 4.76sec, 0.63 ACMR | *(b) Hoppe [1999]* 226ms, 0.65 ACMR | *(c) Our work* 51ms, 0.69 ACMR | *(d) Nehab et al. [2006]* 40sec, 1.21 MOVR, 0.73 ACMR | *(e) Our work* 76ms, 1.18 MOVR, 0.72 ACMR

Figure 1: *Vertex cache efficiency and overdraw. Views of a 40k triangle Dragon model are shown, where red regions represent cache misses, and dark regions represent high overdraw rate. As a preprocessing stage, real-time rendering applications optimize the order triangles are issued to reduce the average post-transform vertex cache miss ratio (ACMR) (a-c). Recent algorithms also minimize the overdraw ratios (OVR) (d-e) with little cache degradation. We present novel algorithms that result in excellent vertex cache efficiency (c) as well as low overdraw (e). Our methods are significantly faster than previous approaches (compare timings), and are suitable for run-time execution.*

Abstract

We present novel algorithms that optimize the order in which triangles are rendered, to improve post-transform vertex cache efficiency as well as for view-independent overdraw reduction. The resulting triangle orders perform on par with previous methods, but are orders magnitude faster to compute.

The improvements in processing speed allow us to perform the optimization right after a model is loaded, when more information on the host hardware is available. This allows our vertex cache optimization to often outperform other methods. In fact, our algorithms can even be executed interactively, allowing for re-optimization in case of changes to geometry or topology, which happen often in CAD/CAM applications. We believe that most real-time rendering applications will immediately benefit from these new results.

1 Introduction

Modern rendering pipelines accept input in a variety of formats, but the most widely used representation for geometry is based on *vertex* and *index buffers*. A vertex buffer provides the 3D coordinates and attributes for a set of vertices. The index buffer defines a set of triangles, each given by the triad of indices of its vertices in the vertex buffer.

As each triangle is processed for rendering, referenced vertices are processed by a *vertex shader* in an operation that can be computationally expensive. The cost comes from a combination of the bandwidth required to load the data associated to each vertex (i.e., position, normal, color, texture coordinates, etc), and the instructions required to process them (i.e., transform and lighting). Applications whose rendering cost is dominated by this bottleneck are said to be *vertex-bound*.

Another potential bottleneck exists during rasterization. Each generated pixel is processed by a *pixel shader*, which might perform expensive operations in the process of computing the final pixel color. Once again, the cost comes from bandwidth associated to texture lookups and from the ALU instructions executed by the GPU. When this cost dominates the total rendering cost, applications are said to be *pixel-bound* (or *fill-bound*). The growing complexity of per-pixel lighting effects has progressively increased concerns with this bottleneck.

Naturally, modern GPUs employ a variety of optimizations that attempt to avoid unnecessary computations and memory references. Two such optimizations are the *post-transform vertex cache*, used during vertex processing, and *early Z-culling*, used during pixel processing. Interestingly, the effectiveness of both optimizations is highly dependent on the order on which triangles are issued. Given the new unified shader architectures, reducing vertex and pixel load at run-time is beneficial for both vertex- and pixel-bound applications. It is therefore desirable to generate a triangle order that is suitable for both optimizations.

The post-transform vertex cache holds a small number of transformed vertices in a FIFO queue. When a triangle references a vertex found in the cache, results are reused directly, without any external data transfers or further processing required. The average cache miss ratio (ACMR) can be greatly reduced if triangles are ordered to increase vertex reference locality. The ACMR in turn has a strong impact on the frame rate of vertex-bound applications (see figure 2a). Many algorithms have therefore been proposed to generate low-ACMR triangle orders (section 2).

Early Z-culling is an optimization by which the GPU tests the depth of each pixel against the Z-buffer before executing its pixel shader. If the depth is such that the results would be discarded, no additional work is performed. This optimization is most effective when there is little *overdraw*. Overdraw can be defined as the ratio between the total number of pixels passing the depth test and the number of visible pixels (a ratio of 1 means no overdraw). The graph in figure 2b illustrates the dependency between overdraw ratios and rendering times, in a pixel-bound scenario.

ACM Reference Format
Sander, P., Nehab, D., Barczak, J. 2007. Fast Triangle Reordering for Vertex Locality and Reduced Overdraw. *ACM Trans. Graph. 26*, 3, Article 89 (July 2007), 9 pages. DOI = 10.1145/1239451.1239540 http://doi.acm.org/10.1145/1239451.1239540.

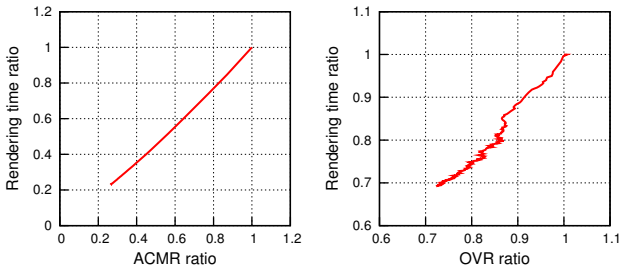


Figure 2: *ACMR and overdraw impact on rendering times. (a) In a completely vertex-bound scenario, the dependency between ACMR and rendering time is almost linear. (b) The same is true of overdraw ratios and rendering times, when pixel shading is the bottleneck (although our measurements were more noisy).*

The task of determining a static triangle order that produces low overdraw without harming the vertex cache locality has been the topic of recent research [Nehab et al. 2006]. The basic insight is to partition each model into triangle clusters, or contiguous patches. Since clusters contain many triangles, preserving vertex locality within clusters is enough to guarantee a low overall ACMR. These clusters can then be atomically sorted according to a metric that minimizes overdraw. Surprisingly, it is possible to design view-independent metrics that result in a static cluster order that greatly minimizes overdraw. The facts that vertex cache performance is not harmed, and that nothing else must be changed in a rendering pipeline, make the idea very attractive.

In this paper, we present novel, extremely efficient algorithms for post-transform vertex cache optimization (section 3), for splitting meshes into triangle clusters that do not significantly worsen the ACMR (section 4), and for sorting these clusters into a view independent order that reduces overdraw (section 5). The main features are the following:

- Our methods are orders of magnitude faster than previous proposals, and thus can be executed interactively;
- As far as quality is concerned, our methods perform on par with previous methods;
- In order to simplify integration with existing applications, our methods operate directly on vertex and index buffers;
- Therefore, unlike many previous methods, our algorithms operate transparently on non-manifold models;
- They are also substantially simpler to implement than most previous methods.

In several application domains, such as games and CAD/CAM applications, models frequently undergo changes in geometry and topology. It is desirable to re-optimize such models, and in that case the efficiency of the optimization process is very important. Our work allows these systems to bring the optimization stage to run-time. At run-time, the exact post-transform vertex cache size might be available, and we can therefore target the correct value. Alternatively, when used as a pre-processing stage, our algorithms can greatly reduce processing time. For these reasons, we believe that many real-time rendering applications, vertex- or pixel-bound, will immediately benefit from our results.

2 Previous work

Vertex cache optimization The first practical approaches to tackle the vertex bandwidth problem were based on triangle strips [Akeley et al. 1990; Evans et al. 1996; Speckmann and Snoeyink 1997; Xiang et al. 1999]. Even early GPUs supported strips natively (with a 2-entry cache), so these methods resulted in ACMR values close to 1, as long as the mesh could be encoded with relatively few strips. Substantial effort was therefore dedicated to the NP-complete problem of finding single-strip representations for meshes [Garey et al. 1976; Dillencourt 1996; Arkin et al. 1996; Estkowski et al. 2002].

In order to reduce bandwidth even further, Deering [1995] proposed a geometry compression scheme based on *generalized triangle meshes*. What set Deering's work apart from other approaches to geometry compression [Taubin and Rossignac 1998; Gumhold and Straßer 1998; Touma and Gotsman 1998] was that the decompression stage was designed to be easily implemented on hardware [Deering and Nelson 1993].

Generalized triangle meshes explicitly manage a fixed size cache, with instructions to replace cached vertices or to reference them to define triangles. Chow [1997] later provided algorithms to encode triangle meshes into this representation, achieving near optimal results (ACMR 0.6–0.7). Bar-Yehuda and Gotsman [1996] proved that for a cache size of k, the optimal ACMR was bound by $0.5+\Omega(1/k)$, whereas to ensure each vertex in a n-vertex mesh is processed only once, a cache size of $O(\sqrt{n})$ was required. Other encodings reaching near optimal ACMR were proposed that break away from strip-like structures [Mitra and Chiueh 1998], but hardware trends followed the path of simplicity.

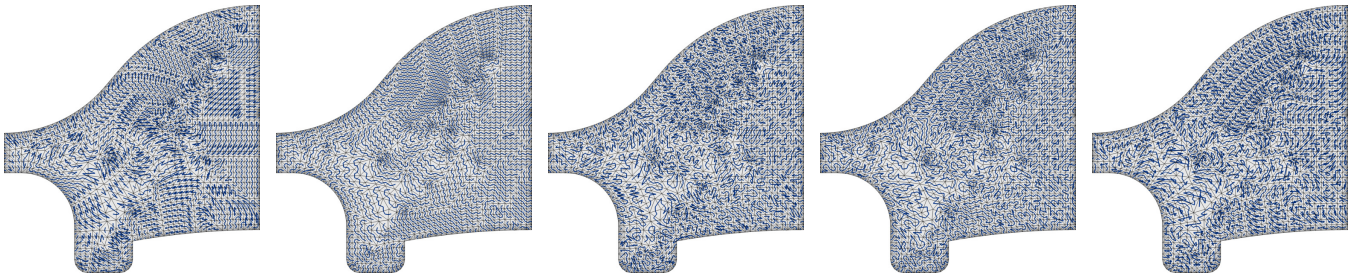
This was mostly due to the pragmatic approach proposed by Hoppe [1999]. He noticed that a FIFO cache, without any explicit management (i.e., transparent), was enough to match the results reported by Chow. This was possible by maximizing vertex locality with the use of parallel short strips. Besides being easier to implement on hardware, his approach was tailored to match established graphics APIs, and therefore modern GPUs include a post transform vertex cache in this fashion.

The use of index buffers to specify connectivity further changed the panorama. It is true that longer strips minimize the length of the index buffer. However, each index takes a small fraction of the amount of information required by each vertex (i.e., position, normal, color, texture coordinates, etc). Hence, the vast majority of memory bandwidth is used on random access to vertex data. Moreover, a considerable share of rendering time is spent processing vertex data, which can only be reduced with higher cache hit rates. In this context, minimizing the ACMR is much more important than compressing the connectivity information. Nevertheless, with high-performance rendering in mind, substantial effort was invested on generating single-strip representations for triangle meshes [Gopi 2004; Diaz-Gutierrez et al. 2006], or on minimizing the number of strips [van Kaick et al. 2004].

Many related methods were also designed that preserve a certain level of locality in dynamic meshes, for applications such as progressive refinement or view-dependent rendering [El-Sana et al. 1999; Velho et al. 1999; Ribelles et al. 2000; Belmonte et al. 2001; Stewart 2001; Shafae and Pajarola 2003; Ramos and Chover 2004; Ripollés et al. 2005]. Instead, we focus on making our method as fast as possible, so it can be executed at run-time.

Lin and Yu [2006] were the first to propose a practical approach that results in lower ACMR than Hoppe [1999]. Their work is also the most closely related to ours. Neither method attempts to generate triangle strips, or perform any type of global analysis or optimization. Triangles are issued based on a greedy, local strategy, based on a simulation of the cache, and driven by vertex adjacency. Therefore, both methods work transparently on non-manifolds. Although their results are slightly better than ours (by about 5–10%), their method takes orders of magnitude longer than ours to optimize the same input (often 100 times longer).

Alternative orderings that are oblivious to cache sizes have been recently proposed. These are driven by the multi-resolution locality properties of space-filling curves [Bogomjakov and Gotsman 2002], or by the minimization of objective functionals that push the triangle ordering in that direction [Yoon et al. 2005]. Although use of these methods is advantageous when cache size is not known a priori, or in combination with progressive meshes, given a reason-



(a) K-Cache-Reorder Lin and Yu [2006] *(b) D3DXMesh based on [Hoppe 1999]* *(c) OpenCCL Yoon and Lindstrom [2006]* *(d) dfsrendseq Bogomjakov et al. [2001]* *(e) Our work*

Figure 3: *Triangle ordering paths. The figures show the patterns generated by a variety of different triangle order optimization algorithms. Adjacent triangles are connected by a blue line, so that the paths are visible. The structure in the triangle orders allows us to generate satisfactory clusters by simply breaking the optimized sequence into smaller, contiguous subsequences of triangles.*

able estimate of the cache size, other methods perform much better (including our own) [Hoppe 1999; Lin and Yu 2006].

Nevertheless, the vertex cache size does vary with the hardware, and could potentially depend on the amount of information being passed from the vertex shader to the pixel shader (see figure 10). In order to perform the optimization during a pre-process stage, Lin and Yu [2006], and Hoppe [1999] must therefore underestimate this value. The efficiency of our method allows us to bring the optimization to run-time, and this gives our method an edge over previous alternatives. Finally, we can also re-optimize models to reflect changes, and this makes our method especially useful for CAD/CAM applications.

Overdraw reduction A common approach to overdraw reduction exploits early Z-culling with two rendering passes. On the first pass, using no pixel shader, the geometry is rendered to prime the Z-buffer. On the second pass, pixel shading is enabled, and the scene is rendered again, except this time the depth test is changed to *less then or equal*. Since only visible pixel will pass that test, the expensive shading computations will only be executed on them. For pixel intensive applications, it might be well worth it to render geometry twice in order to reduce the pixel load. However, when geometry is also complex, this solution is not acceptable.

In such cases, visibility sorting and occlusion culling techniques [Airey 1990; Teller and Sèquin 1991; Greene et al. 1993] are recommended. In order to determine visibility, modern methods [Hillesland et al. 2002; Bittner et al. 2004; Govindaraju et al. 2005] use hardware-based occlusion queries, and some can take advantage of predicated rendering [Blythe 2006]. Although these methods can reduce overdraw, they traditionally operate at coarse levels, grouping primitives together. Our overdraw reduction strategy could therefore be used to find an appropriate order in which to render the primitives in each group that passes the occlusion test.

Interestingly, in the presence of early Z-culling, the amount of overdraw is determined by the order on which the primitives are traversed. When predicated rendering is used, a group can only be culled if none of its pixels passes the depth test. In that case, although rendering the primitives would certainly be wasteful, the action would not incur any additional overdraw. Furthermore, due to the use of proxy geometry in occlusion queries, the Z-buffer bandwidth would not change significantly either.

As far as ordering is concerned, the best strategy is to draw triangles in front-to-back order. The most direct way to achieve this is to use view-dependent sorting. Although there are efficient approaches to visibility sorting [Govindaraju et al. 2005], it would be convenient to find a view-independent order that reduces overdraw without requiring any changes to the application.

This is the problem addressed by Nehab et al. [2006]. To produce such an ordering, they cluster the mesh into planar patches, and sort the resulting patches by approximating a solution to the minimum feedback arc set problem on the partial order graph that represents pairwise cluster occlusions. Each patch can later be independently optimized for vertex cache efficiency, using any method of choice. Since the patches include a considerable number of triangles, the technique does not harm vertex bound applications.

This method was also designed to operate off-line, and little consideration was given to running times. In fact, the clustering and partial order graph generation stages can each take in the order of *minutes* to complete. Our new method follows a different strategy. We design a metric that is extremely efficient to evaluate. This allows us to use a much larger number of clusters, which in turn causes their shape to be less relevant. We can therefore design a fast clustering method. In order to prevent these smaller clusters from harming the vertex cache, we generate them during the vertex cache optimization itself, taking into account the state of the cache.

3 Post-transform vertex cache optimization

Vertex and index buffers provide no explicit adjacency information. Worst of all, models so defined can contain multiple disconnected components and even represent non-manifold meshes. Triangle adjacency information (the dual graph), generally required by strip-based methods, can be awkward to maintain for non-manifolds. We are looking for an algorithm that runs to completion in a time budget comparable to that of generating this information. We therefore restrict ourselves to considering only *vertex-triangle* adjacency. This information is efficiently captured by an offset map into an array that lists the triangles for each vertex.

Basic idea Given this vertex adjacency, we could directly output the triangle lists for each vertex (issuing each triangle only once, of course). Intuitively, the operation clusters triangles around common vertices, much like triangle fans. However, triangles are ordered randomly within each fan (see figure 4a). Interestingly, within a small neighborhood, the relative order on which the triangles are generated does not matter, as long as the post-transform vertex cache is large enough to hold the entire neighborhood (i.e., 7 vertices in regular meshes). We therefore do not have to sort the triangles before issuing them. This gives us the freedom to explore triangle orders that are locally random or, in other words, *tipsy*.

When considering variants of this idea, it is hard to predict the expected cache hit ratio, especially in non-uniform models. Nevertheless, we can gain valuable insight by analyzing the algorithm under certain simplifying assumptions, which we call *steady state*. For that, assume a large cache size, and an infinite, regular tessellation. For the example in figure 4a, it is easy to see that the steady state ACMR is $7/6$. Naturally, in real models, the final ACMR will be worse. This is due to the fact that triangle fans will start colliding, and at some point we will be forced to reference $n+1$ vertices, while issuing only n triangles, for $n < 6$. Fortunately, we can do much better than $\frac{n+1}{n}$ by carefully selecting the order on which vertices are *fanned* (i.e., their triangles are emitted).

Fanning vertex sequence While a vertex is being fanned, each issued triangle references other vertices. These form a subset of the 1-ring of the fanning vertex. For efficiency reasons, we select the next fanning vertex from among this subset, which we call the *1-ring candidates*. Among them, we consider those that would still be in the cache, even after being fanned themselves. If there are multiple options, we pick the candidate that entered the cache the earliest. Otherwise, we arbitrarily pick the first 1-ring candidate.

When the algorithm reaches a point where all 1-ring candidates have already been fanned (i.e., a *dead-end*), we select the most recently referenced vertex that still has non-issued triangles. In many cases, this vertex is still in the cache. Therefore, this choice is better than picking an arbitrary vertex based on the input order alone, which we do only as a last resort.

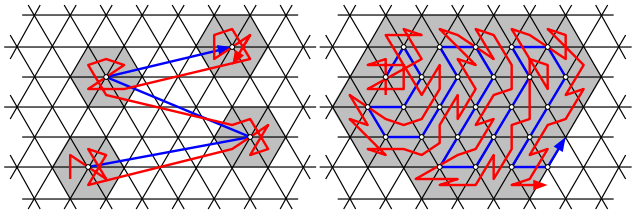
The linear time algorithm The complete pseudo-code for the algorithm is presented below, followed by a discussion.

```
Tipsify(I, k): O
 A = Build-Adjacency(I)          Vertex-triangle adjacency
 L = Get-Triangle-Counts(A)      Per-vertex live triangle counts
 C = Zero(Vertex-Count(I))       Per-vertex caching time stamps
 D = Empty-Stack()               Dead-end vertex stack
 E = False(Triangle-Count(I))    Per triangle emitted flag
 O = Empty-Index-Buffer()        Empty output buffer
 f = 0                           Arbitrary starting vertex
 s = k+1, i = 1                  Time stamp and cursor
 while f >= 0                    For all valid fanning vertices
   N = Empty-Set()               1-ring of next candidates
   foreach Triangle t in Neighbors(A, f)
     if !Emitted(E,t)
       for each Vertex v in t
         Append(O,v)             Output vertex
         Push(D,v)               Add to dead-end stack
         Insert(N,v)             Register as candidate
         L[v] = L[v]-1           Decrease live triangle count
         if s-C[v] > k           If not in cache
           C[v] = s              Set time stamp
           s = s+1               Increment time stamp
       E[t] = true               Flag triangle as emitted
   Select next fanning vertex
   f = Get-Next-Vertex(I,i,k,N,C,s,L,D)
 return O

Get-Next-Vertex(I,i,k,N,C,s,L,D)
 n = -1, p = -1                  Best candidate and priority
 foreach Vertex v in N
   if L[v] > 0                   Must have live triangles
      p = 0                      Initial priority
      if s-C[v]+2*L[v] <= k      In cache even after fanning?
         p = s-C[v]              Priority is position in cache
      if p > m                   Keep best candidate
         m = p
         n = v
 if n == -1                      Reached a dead-end?
   n = Skip-Dead-End(L,D,I,i)    Get non-local vertex
 return n

Skip-Dead-End(L,D,I,i)
 while !Empty(D)                 Next in dead-end stack
   d = Pop(D)
   if L[d] > 0                   Check for live triangles
      return d
 while i < Vertex-Count(I)       Next in input order
   i = i + 1                     Cursor sweeps list only once
   if L[i] > 0                   Check for live triangles
      return i
 return -1                       We are done!
```

The function `Tipsify()` receives an index buffer as input, and outputs an optimized index buffer containing the same triangles, reordered according to the algorithm we outlined.



(a) Tipsy fans *(b) Tipsify*

Figure 4: *Tipsification. The red line shows the order on which the triangles are issued. The blue line shows the fanning vertex sequence. (a) Randomly choosing the fanning vertices yields 7/6 ACMR. (b) With the zig zag pattern of period 2n, we reach the nearly optimal steady state of* $\frac{n+2}{2n+2} \rightarrow 0.5$ *ACMR.*

The first step in the algorithm is to build the sets of all triangles adjacent to each vertex. `Build-Adjacency()` can be efficiently implemented with three linear passes over the index buffer (much like the counting-sort algorithm). On the first pass, we count the number of occurrences of each vertex. Then, with a pass over the counted values, a running sum is computed that produces the offset map. The array with the triangle lists can finally be produced with another pass over the index buffer.

`Tipsify()` also uses an array `L` that maintains, for each vertex, the number adjacent *live triangles*, i.e., the number of neighboring triangles that have not yet been written to the output buffer. `Init-Live-Triangles()` initializes this array directly from the adjacency structure `A`.

The remaining required data structures are very simple. The array `C` holds, for each vertex, the time at which it last entered the cache. The concept of time is given by a time-stamp counter `s` that is incremented each time a vertex enters the cache. Given this information and the current time stamp, we can compute in constant-time the position of a vertex `v` in the FIFO cache. Simply subtract both numbers to get `s-C[v]`. The dead-end stack `D` helps the algorithm recover from the absence of good next candidates within the 1-ring of the current fanning vertex. Finally, the array `E` flags triangles that have already been emitted.

After arbitrarily initializing the fanning vertex `f` to the first input vertex, the algorithm enters its main loop. This loop will eventually issue all triangles in the input, by selecting an efficient sequence of fanning vertices. The loop itself is very simple, and only takes care of minor bookkeeping. The interesting part of the algorithm is captured by the function `Get-Next-Vertex()`.

`Get-Next-Vertex()` considers all 1-ring candidates and selects the best next fanning vertex according to our metric. Naturally, a good candidate must have live triangles, and only those are considered. Next, we ignore vertices that would not be in the cache after being fanned themselves. Instead of looking ahead to decide which candidates fall within this category, we rely on a constant-time, conservative approximation.

Recall the position of a vertex `v` in the cache is given by the expression `s-C[v]`. We also know that `v` has `L[v]` live triangles. If we were to fan this vertex, each triangle would at most generate two cache misses. This would cause `v` to be shifted into position `s-C[v]+2*L[v]` in the cache. Therefore, to remain in the cache it is sufficient that `s-C[v]+2*L[v] > k`. Of the candidates passing this test, we choose the one that has entered the cache the earliest, because it is still useful, but is about to be evicted.

When there are no suitable candidates, we have reached a dead-end and might have to jump to a non-local vertex. This is the task of the `Skip-Dead-End()` function. To increase the chances of restarting from a vertex that is still in the cache, we keep a stack `D` of recently issued vertices. This allows us to efficiently search among these vertices, in reverse order, until we find a vertex with live triangles. Finally, if we exhaust the dead-end stack without producing

a fanning vertex, we simply take the next vertex with live triangles, in input order.

Amortized run-time analysis Unlike most previous approaches, our method runs in time that is linear on the input size. The running time does not depend on the target cache size `k`. This is clear from the fact that `k` only appears in constant-time expressions.

For the run-time analysis, it is easier to initially exclude the cost of `Get-Next-Vertex()`. In that case, the main loop in `Tipsify()` runs in time $O(t)$, where t is the number of input triangles. Each vertex is fanned at most once, and each fanning operation only visits the vertex's neighboring triangles. Therefore, each triangle is visited at most three times (one for each of its vertices). Furthermore, for each visited triangle, all operations are constant-time.

As for `Get-Next-Vertex()`, notice that along its entire lifetime, the dead-end stack `D` receives only $3t$ indices. This is because each triangle is emitted only once, and each triangle pushes only its three vertex indices on `D`. Therefore, the first loop in `Get-Next-Vertex()` can only be executed $3t$ times. Next, consider the second loop. Index `i` gets incremented, but it is never decremented. Therefore, this loop also can only be executed $3t$ times, and this completes the proof.

Steady state analysis Interestingly, after an initial spiraling pattern, our method converges to a zig zag pattern, as shown in figure 4b. Intuitively, this zig zag occurs because the fanning sequence tries to follow the previous strip of vertices, since they are still in the cache (remember the algorithm encourages fanning vertices that have entered the cache earlier). Eventually, it reaches a point where the adjacent vertices from the preceding strip are not on the cache. The sequence is then forced to turn around again in order to fan a recently processed vertex that is still in the cache.

Counting the number of issued triangles versus the number of newly transformed vertices at each cycle, we reach the steady state performance of $\frac{n+2}{2n+2}$ ACMR. Whenever $n > 2$, this improves on the $7/6$ *ACMR* we had before. Furthermore, the larger the n, the closer we get to the optimal value of $1/2$.

4 Fast linear clustering

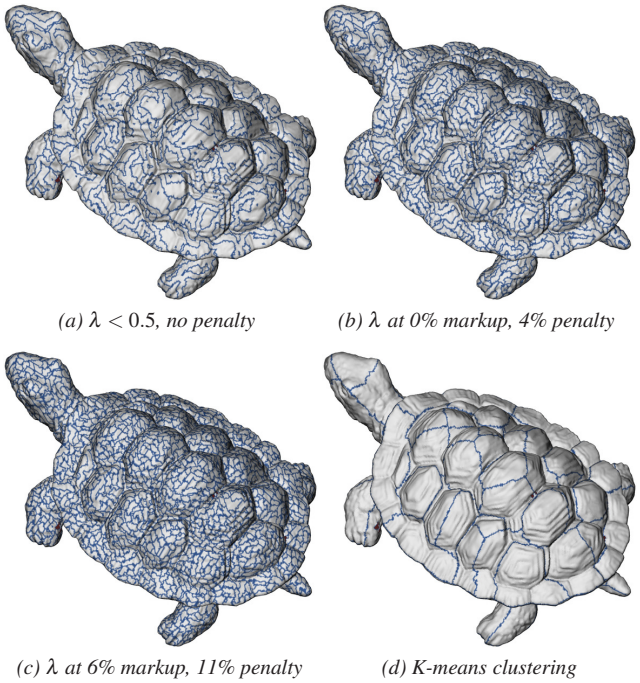
The larger the clusters, the smaller the impact on ACMR when we later change the relative order between them. Unfortunately, given only a few large clusters, the ordering stage might not have enough freedom to reduce overdraw. Hence, there is a trade-off. Our approach is to take advantage of the wealth of information produced during the vertex cache optimization process, and break the model into a multitude of small clusters, carefully delimited not to significantly penalize the vertex cache.

Due to locality requirements, the sequence of triangles issued by most vertex cache optimization algorithms follows wide paths along the mesh (see figure 3). Naturally, triangles issued sequentially tend to be spatially adjacent. Given this structure, we can reduce the clustering stage from a 2-dimensional problem to a 1-dimensional problem. We simply break the output of the vertex cache optimization into contiguous, smaller subsequences, in a process we call *fast linear clustering*. Each subsequence then becomes a cluster.

In the case of our algorithm, each dead-end results in a path discontinuity. These discontinuities naturally cause the vertex cache to be flushed, and we can therefore safely break the sequence at these *hard boundaries*. Nevertheless, in the absence of hard boundaries, long continuous paths must still be broken into smaller pieces. This is done with the addition of *soft boundaries*.

Assume we are scanning a contiguous sequence of triangles and must decide whether to place a soft boundary before triangle i. Our algorithm breaks the sequence when following expression is true:

$$M_i < \lambda \qquad (1)$$



(a) $\lambda < 0.5$, no penalty *(b) λ at 0% markup, 4% penalty*

(c) λ at 6% markup, 11% penalty *(d) K-means clustering*

Figure 5: *The effect of λ. (a) Setting λ lower than the global ACMR results in breaks only on hard boundaries. (b-c) Reducing λ produces more clusters, but results in higher losses to vertex processing. (d) In contrast, Nehab et al. [2006] use fewer, larger clusters.*

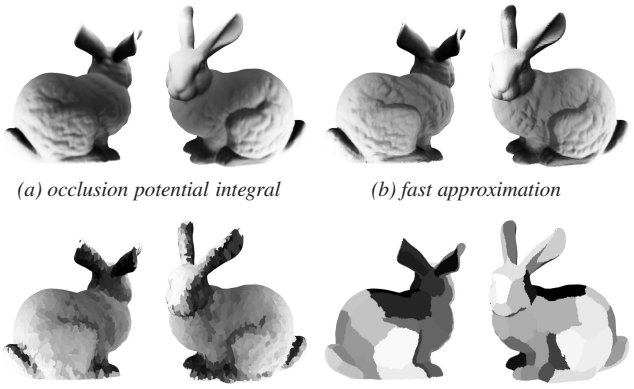
M_i is the local ACMR for the current subsequence, and therefore decreases from 3 towards 0.5. This estimate is available nearly for free during the vertex cache optimization process, so the clustering process is extremely efficient.

Smaller subsequences, which have only recently paid the price of a complete cache flush, have a higher local ACMR M_i, and are therefore less likely to be broken off. Higher values of λ allow even these sequences to be broken, and result in a greater number of clusters, at the cost of a stronger impact on the global ACMR. Conversely, lower values of λ reduce the number of soft boundaries, and thus affect the global ACMR to a lesser extent. In fact, λ is a close lower bound for the ACMR of the clustered model. In particular, choosing $\lambda < 0.5$ eliminates soft boundaries and leaves the ACMR unchanged. Any value $\lambda \geq 3$ will generate individual triangles as clusters, potentially ruining the ACMR.

Even before clustering, the ACMR can vary substantially between different models (see figure 9). It therefore makes sense to choose λ relative to the ACMR each model has right before clustering affects it. This allows us to bound the penalty in vertex processing due to clustering. We have found that, for a variety of models, a markup of only $< 5\%$ in the value of λ results in a sufficient number of clusters, while incurring a small ($\sim$5–10%) penalty in vertex processing.

Figure 5 shows examples of the Turtle model clustered by our algorithm, with different values of λ. As expected, increasing the value of λ results in more clusters, and has a higher impact on ACMR. Also as expected, comparing with the results of the k-means chartification used by Nehab et al. [2006], the fast linear clustering method results in a much larger number of smaller clusters. This in turn allows us to use a view-independent metric that simply assigns a numeric rank to each cluster. We can then sort the clusters with regard to its rank, which is at the same time very effective and extremely efficient to compute.

The clustering stage also substantially reduces the number of elements to be sorted. In fact, the $O(n \log n)$ complexity is now for n clusters, not t triangles. As long as the average cluster size is



(a) occlusion potential integral *(b) fast approximation*

(c) fast approximation on clusters *(d) Nehab et al. [2006]*

Figure 6: *View independent triangle orders. Bright regions should be drawn first. (a) The occlusion potential integral (equation 2). (b-c) The fast approximation of equation 4, at the triangle level (b) and at the cluster level (c). Notice how the approximation respects the relative order dictated by the integral. (d) Unsurprisingly, Nehab et al. [2006] produces a similar order, although at a much coarser level.*

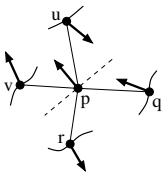
greater than $\log t$, the complexity is sub-linear on the number of triangles. In practice, this is always the case. For instance, even with a tiny average cluster size of 20 (which would definitely harm the vertex cache performance), it would take a model with $t > 20 \cdot 2^{20}$ triangles before $n \log n > t$.

5 View-independent cluster sorting

To minimize overdraw in a view-independent way, we should start by drawing surface points that are more likely to occlude other surface points, from any viewpoint. This likelihood is captured by the *occlusion potential integral* $\mathcal{O}(p,M)$ of an oriented point p, relative to model M. It is defined as the area of M that can be occluded by point p, and is given by the following equation:

$$\mathcal{O}(p,M) = \int_{q \in M} \frac{R(\langle p-q, n_p \rangle) R(\langle p-q, n_q \rangle)}{\langle p-q, p-q \rangle} \, \mathrm{d}q \qquad (2)$$

where $R(x) = \frac{x+|x|}{2}$ is the unit ramp function, $\langle , \rangle$ is the dot product, and n_p and n_q are the normals at points p and q, respectively.



In order to understand the integral, consider the diagram on the left. A point p can only occlude a point q if neither of them is back-face culled when p is in front of q. In other words, $\langle p-q, n_p \rangle$ and $\langle p-q, n_q \rangle$ must both be positive. In the diagram, points r, u, and v fail one or both these tests, and therefore do not contribute to p's occlusion potential. Point q passes both tests. In that case, the contribution reflects the foreshortening of both p and q, as seen by an orthographic camera. This is the role of the cosine terms arising from the dot products and normalized by the denominator.

Ideally, we would like to sort individual surface points based on their occlusion potentials (points with higher potentials drawn first). However, to preserve the vertex cache, we must instead sort triangle clusters atomically. We therefore extend the definition of occlusion potential to surface patches P, as follows:

$$\mathcal{O}(P,M) = \int_{p \in P} \mathcal{O}(p,M) \, \mathrm{d}p \qquad (3)$$

Unfortunately, computing the occlusion potential integral takes $O(t^2)$ time, where t is the number of triangles in the mesh. Instead, we resort to an $\mathrm{O}(t)$ approximation that can be efficiently calculated. Intuitively, points with high occlusion potential will be on the outskirts of a model, and will be pointing away from it. We therefore define the *approximate occlusion potential* of a surface patch P with regard to model M, as

$$\mathcal{O}'(P,M) = (C(P) - C(M)) \cdot N(P) \qquad (4)$$

where C is the centroid function, and $N(P)$ represents the average normal of patch P.

In general, the value of the approximate occlusion potential will not be close to the value of the integral. In fact, the approximation can produce negative values. Nevertheless, the relative order of the values produced tend to be very similar. Figure 6a shows the Bunny model, on which each vertex has been shaded according to equation 2. The vertices were sorted, and then colored from black to white in decreasing order of occlusion potential. Figure 6b was instead ordered and shaded according to equation 4 (at the triangle level). Notice the dark regions between the ears, and also above the tail, paws, and thighs. These regions do not occlude any part of the model, regardless of viewpoint, and are therefore drawn last. Conversely, the top of the head cannot be occluded, and is therefore drawn first. The same approximation can be performed at the cluster level. Figures 6c shows the results of sorting the fast linear clusters described in the previous section. The clusters are so small that results are similar to the triangle level sorting. Comparing with figure 6d, we can see that the overall order followed by Nehab et al. [2006] is similar, although at a much coarser level.

6 Results

Running times Figure 7 (left) shows the running times, in logarithmic scale, of `Tipsify()` and several other vertex cache optimization methods. Tests were run on an Intel Pentium 4 2GHz processor. The Dragon model was decimated into 20 uniformly spaced resolutions ranging from 40k to 800k triangles. A variety of algorithms (the original authors' implementations) were used to optimize each model for vertex locality. Note that our method is considerably faster than all other methods, and runs approximately 100x faster than K-Cache-Reorder, which produces the best triangle orders. The ratios would be even larger when optimizing for large cache sizes (the plot uses a cache size of 12), since our running time depends only on the number of triangles. Figure 7 (right) compares the running times (also in logarithmic scale) of our entire system with that of Nehab et al. [2006], which is the only previous method that optimizes for vertex cache and overdraw. In this case, our new approach is approximately 1,000 times faster.

Due to the significant reduction in processing time, which allows us to optimize models at load time, we can take advantage of the actual host hardware post-transform vertex cache sizes and the amount of information stored per vertex to generate a triangle order optimized for these particular settings. Until this information is directly available through official APIs, the best parameters for each configuration can be tabulated and looked up at run-time.

Vertex cache optimization To avoid cache flushes, previous methods must optimize for conservative cache sizes (K-Cache-Reorder, based on [Lin and Yu 2006] and D3DXMesh, based on [Hoppe 1999]), and yield suboptimal performance in systems with a large cache size. The graphs in figure 9 show that K-Cache-Reorder yields the best ACMR results when optimizing for cache size 12. However, if the run-time cache size increases above 20-24, models optimized at load time with `Tipsify()` can achieve even better ACMRs. In contrast, cache oblivious methods (dfsrenseq [Bogomjakov and Gotsman 2002] and OpenCCL [Yoon and Lindstrom 2006]) only surpass K-Cache-Reorder at much larger cache sizes.

To verify that improvements in ACMR translate into higher frame rates, we tested a vertex bound application on different hardware. We also varied vertex format configurations, since larger vertices (i.e., with more attributes) consume more cache space, and can reduce the number of entries available. In each test, a model was optimized with K-Cache-Reorder, assuming cache sizes 12 (rec-

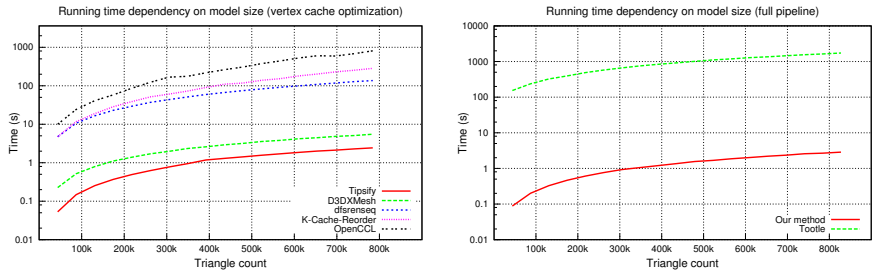


Figure 7: *Running times as a function of model size (in logarithmic scale). The Dragon model was decimated into 20 uniformly spaced resolutions ranging from 40k to 800k triangles. (Left) Vertex cache optimization only. (Right) Running times including the overdraw reduction stage. Notice our methods are much faster than the alternatives.*

ommended by ATI) and 24 (recommended by NVIDIA). The same model was then optimized with `Tipsify()` at run time, given a variety of different cache sizes. Figures 10 and 11 show the rendering time ratios between the models optimized with K-Cache-Reorder and the models optimized with `Tipsify()`, so that ratios above one indicate that `Tipsify()` should be used instead.

Figure 11 shows the results for the ATI X1900XTX graphics card. Regardless of vertex format, results suggest that the effective cache size is 12. Accordingly, the best alternative is to use K-Cache-Reorder with this cache size. However, if the model was optimized with K-Cache-Reorder for cache size 24, our method can result in frame rate improvements of approximately 30%.

Interestingly, both the NVIDIA 7900GTX and 8800GTX seem to have larger caches (see figure 10). Consider the 7900GTX (top row). In nearly all cases, `Tipsify()` produces the greatest improvements by optimizing for a very large cache size. For a vertex output size of 2 to 7 vector attributes, it consistently peaks when optimizing for a cache size of 44–48. As the number of vector attributes increases, we notice a decrease in the optimal choice for cache size.

On the 8800GTX (bottom row), the peak occurs at lower cache size values, and we get little gain by re-optimizing at run-time. In fact, with few vector attributes, surprisingly any value between 12 and 24 yield satisfactory results. A notable exception happens when many attributes are used (9 or more), which seems to cause the card to support only 12 cache entries. In this case, we can improve rendering times by nearly 30% when compared against K-Cache-Reorder optimizing for cache size of 24, which is recommended by NVIDIA.

As shown in the above analysis, if the application is aware of the GPU type and vertex format configuration, `Tipsify()` can provide competitive results, and often improve rendering time at a very small processing cost.

Overdraw reduction Figure 8 shows that the results of our overdraw reduction algorithm are comparable with those produced by the ATI Tootle library (based on [Nehab et al. 2006]), which requires between 300x-10000x more processing time for the same models. Our algorithm can also trade-off between vertex cache efficiency and overdraw reduction through the λ parameter. Note in figure 8 that as λ increases, the cache efficiency decreases, while overdraw efficiency increases. We have found that by setting λ between 0.7 and 0.8, or to a small fraction higher than the post-`Tipsify()` vertex cache efficiency, we get excellent results for both vertex cache and overdraw. However, if one is particularly concerned with one of the optimizations, results can be be further improved by adjusting λ accordingly.

View-dependent sort strategies that dynamically change the order in which triangles are rendered have the potential to completely eliminate overdraw, although at the expense of additional processing. We therefore compared our view-independent, static cluster sort method against a front-to-back rendering strategy that updates the cluster order when the viewpoint changes. After experimenting

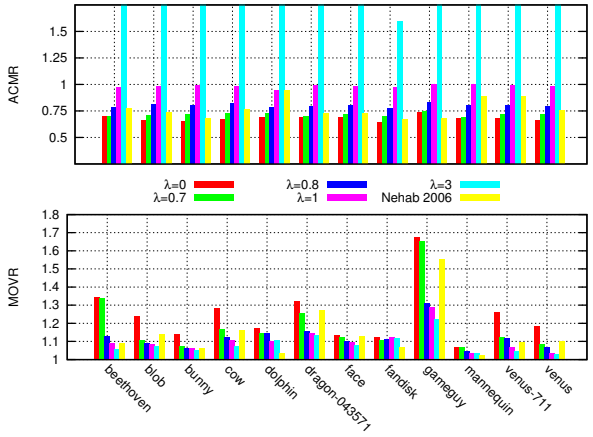


Figure 8: *The trade-off between vertex cache and overdraw. Notice how the choice of λ dictates ACMR penalty directly. If the application can afford a higher ACMR (higher choice of lambda), then overdraw results are lower. Results from Nehab et al. [2006] are also shown for comparison.*

with a variety of models and scenarios, we concluded that dynamic depth sorting is justified when the pixel shader is expensive enough to offset the additional overhead. Using an extremely expensive pixel shader, we were able to obtain a 5% average performance boost on our models by using depth sorting. However, using a simpler shader that performs basic lighting and texturing (although still in a pixel bound scenario), our static sorting strategy beat dynamic sorting by an average of 25%. When the camera is moved further away from the object, pixel load can be reduced arbitrarily and the scenario changes from pixel-bound to vertex-bound. In that case, our method performs only slightly worse than the simple vertex cache optimization, whereas the degradation caused by dynamic depth sorting becomes even more evident.

For highly vertex bound and complex scenes, since the vertex processing component of our algorithm only addresses vertex locality, we recommend using view-dependent level of detail and occlusion culling methods to generate index buffers which can then be processed by our algorithms to further reduce vertex and pixel processing. Note that our methods are completely orthogonal to these approaches, and entirely transparent to the application at rendering time. It takes direct advantage of already available hardware computation culling optimizations.

7 Conclusions

In this paper we introduced algorithms that efficiently reorder triangles in a model to take advantage of graphics hardware optimizations during rendering. Our algorithms are orders of magnitude faster than previous methods for both vertex cache and overdraw optimization. Furthermore, because the algorithms can be executed in a fraction of the time, if we optimize for a the hardware's vertex cache size at load time, we show that the vertex-cache optimization

algorithm alone can often yield better results than previous methods. The overdraw reduction method doesn't significantly harm vertex cache efficiency while achieving comparable overdraw results to previous work. Additionally, the algorithms are general, simple to implement, and easy to integrate with rendering applications. We expect these new methods to be very useful for a wide range of real-time rendering applications, including those that require interactive optimization, such as CAD applications.

Acknowledgements

We would like to thank Phil Rogers of AMD for his suggestions and many discussions on an earlier triangle ordering algorithm, which eventually led us to pursue this work. We also thank the members of AMD's 3D Application Research group for their unyielding support, suggestions, and comments.

References

AIREY, J. M. 1990. *Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations*. PhD thesis, UNC-CH.

AKELEY, K., HAEBERLI, P., and BURNS, D. 1990. The tomesh.c program. Available on SGI computers and developers toolbox CD.

ARKIN, E. M., HELD, M., MITCHELL, J. S. B., and SKIENA, S. 1996. Hamiltonian triangulations for fast rendering. *The Visual Computer*, 12(9):429–444.

BAR-YEHUDA, R. and GOTSMAN, C. 1996. Time/space tradeoffs for polygon mesh rendering. *ACM Transactions on Graphics*, 15 (2):141–152.

BELMONTE, O., REMOLAR, I., J. RIBELLES, M. C., REBOLLO, C., and FERNÁNDEZ, M. 2001. Multiresolution triangle strips. In *Proceedings of IASTED VIIP*, pages 182–187.

BITTNER, J., WIMMER, M., and HARALD PIRINGER, W. P. 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3):615–624.

BLYTHE, D. 2006. The Direct3D 10 system. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2003)*, 25(3):724–734.

BOGOMJAKOV, A. and GOTSMAN, C. 2002. Universal rendering sequences for transparent vertex caching of progressive meshes. *Computer Graphics Forum*, 21(2):137–148.

CHOW, M. M. 1997. Optimized geometry compression for real-time rendering. In *Visualization'97*, pages 347–354, 559.

DEERING, M. 1995. Geometry compression. In *Proc. of ACM SIGGRAPH 95*, pages 13–20.

DEERING, M. F. and NELSON, S. R. 1993. Leo: a system for cost effective 3D shaded graphics. In *Proc. of ACM SIGGRAPH 93*, pages 101–108.

DIAZ-GUTIERREZ, P., BHUSHAN, A., GOPI, M., and PAJAROLA, R. 2006. Single-strips for fast interactive rendering. *The Visual Computer*, 22(6):372–386.

DILLENCOURT, M. B. 1996. Finding hamiltonian cycles in delaunay triangulations is NP-complete. *Discrete Applied Mathematics*, 64(3):207–217.

EL-SANA, J. A., AZANLI, E., and VARSHNEY, A. 1999. Skip strips: maintaining triangle strips for view-dependent rendering. In *Visualization '99*, pages 131–138.

ESTKOWSKI, R., MITCHELL, J., and XI-ANG., X. 2002. Optimal decomposition of polygonal models into triangle strips. In *SCG*, pages 254–263.

EVANS, F., SKIENA, S., and VARSHNEY, A. 1996. Optimizing triangle strips for fast rendering. In *Visualization '96*, pages 319–326.

GAREY, M. R., JOHNSON, D. S., and TARJAN, R. E. 1976. The planar hamiltonian circuit problem is NP-complete. *SIAM J. Comput.*, 5(4):704–714.

GOPI, M. 2004. Controllable single-strip generation for triangulated surfaces. In *PG'04*, pages 61–69.

GOVINDARAJU, N. K., HENSON, M., LIN, M. C., and MANOCHA, D. 2005. Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In *I3D*, pages 49–56.

GREENE, N., KASS, M., and MILLER, G. 1993. Hierarchical z-buffer visibility. In *Proc. of ACM SIGGRAPH 93*, pages 231–238.

GUMHOLD, S. and STRASSER, W. 1998. Real time compression of triangle mesh connectivity. In *Proc. of ACM SIGGRAPH 98*, pages 133–140.

HILLESLAND, K., A., B. S., D., L., and MANOCHA. 2002. Fast and simple occlusion culling using hardware-based depth queries. Technical Report TR02-039, Department of Computer Science, UNC-CH.

HOPPE, H. 1999. Optimization of mesh locality for transparent vertex caching. In *Proc. of ACM SIGGRAPH 99*, pages 269–276.

LIN, G. and YU, T. P.-Y. 2006. An improved vertex caching scheme for 3d mesh rendering. *TVCG*, 12(4):640–648.

MITRA, T. and CHIUEH, T. 1998. A breadth-first approach to efficient mesh traversal. In *HWWS'98*, pages 31–38.

NEHAB, D., BARCZAK, J., and SANDER, P. V. 2006. Triangle order optimization for graphics hardware computation culling. In *I3D*, pages 207–211.

RAMOS, J. and CHOVER, M. 2004. Lodstrips: Level of detail strips. *Lecture Notes in Computer Science*, 3039:107–114.

RIBELLES, J., LÓPEZ, A., REMOLAR, I., BELMONTE, O., and CHOVER, M. 2000. Multiresolution modelling of polygonal surface meshes using triangle fans. *Lecture Notes in Computer Science*, 1953:431–443.

RIPOLLÉS, O., CHOVER, M., and RAMOS, J. F. 2005. Quality strips for models with level of detail. In *Proceedings of IASTED VIIP*, ACTA Press, pages 268–273.

SHAFAE, M. and PAJAROLA, R. 2003. Dstrips: Dynamic triangle strips for real-time mesh simplification and rendering. In *PG'03*, pages 271–280.

SPECKMANN, B. and SNOEYINK, J. 1997. Easy triangle strips for tin terrain models. In *Canadian Conference on Computational Geometry*, pages 91–100.

STEWART, A. J. 2001. Tunneling for triangle strips in continuous level-of-detail meshes. In *Graphics Interface*, pages 91–100.

TAUBIN, G. and ROSSIGNAC, J. 1998. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17 (2):84–115.

TELLER, S. J. and SÈQUIN, C. H. 1991. Visibility preprocessing for interactive walkthroughs. In *Proc. of ACM SIGGRAPH 91*, pages 61–70.

TOUMA, C. and GOTSMAN, C. 1998. Triangle mesh compression. In *Proceedings of Graphics Interface*, pages 26–34.

VAN KAICK, O., DA SILVA, M., and PEDRINI, H. 2004. Efficient generation of triangle strips from triangulated meshes. *Journal of WSCG*, 12(1–3):475–481.

VELHO, L., DE FIGUEIREDO, L. H., and GOMES, J. 1999. Hierarchical generalized triangle strips. *The Visual Computer*, 15(1): 21–35.

XIANG, X., HELD, M., and MITCHELL, J. S. B. 1999. Fast and effective stripification of polygonal surface models. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics*, pages 71–78.

YOON, S.-E. and LINDSTROM, P. 2006. Mesh layouts for block-based caches. *TVCG*, 12(5):1213–1220.

YOON, S.-E., LINDSTROM, P., PASCUCCI, V., and MANOCHA, D. 2005. Cache-oblivious mesh layouts. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2005)*, 24(3):886–893.

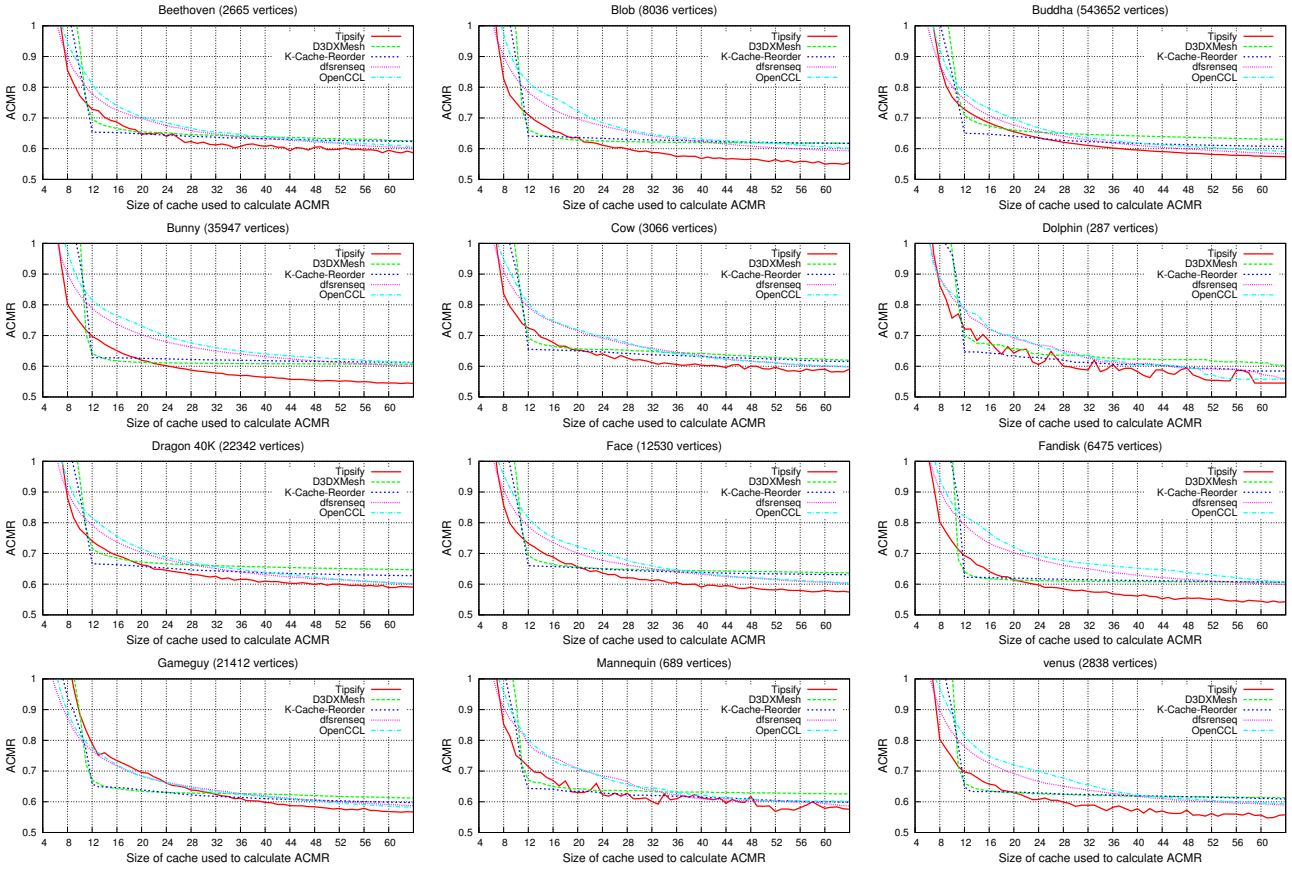


Figure 9: *ACMR comparison of Tipsify against several cache optimization methods (using their respective authors' implementations). Competing methods were optimized for a cache size of 12, at pre-processing. Tipsify was instead given the cache size used to measure ACMR. In order words, the results are shown as if Tipsify was used at run-time.*

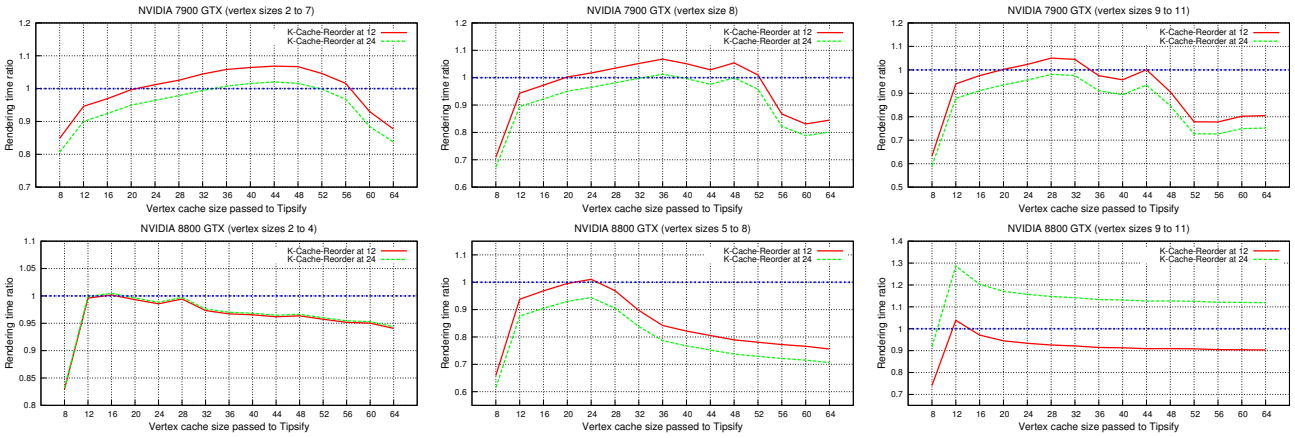


Figure 10: *Rendering time ratio between K-Cache-Reorder (using cache sizes of 12 or 24), and Tipsify (running on a variety of vertex cache sizes). Lines above the 1 mean Tipsify is doing better than K-Cache-Reorder. Plot shows one line for each number of vertex attributes (for simplicity, similar lines are grouped together). Tests were run on two of NVIDIA latest graphics cards.*

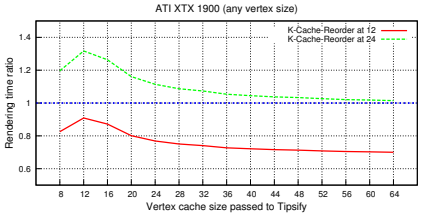


Figure 11: *Same as figure 10, but on the latest ATI hardware.*