Optimization of Mesh Locality for Transparent Vertex Caching

Hugues Hoppe
Microsoft Research

ABSTRACT

Bus traffic between the graphics subsystem and memory can become a bottleneck when rendering geometrically complex meshes. In this paper, we investigate the use of vertex caching to transparently reduce geometry bandwidth. Use of an indexed triangle strip representation permits application programs to animate the meshes at video rates, and provides backward compatibility on legacy hardware. The efficiency of vertex caching is maximized by reordering the faces in the mesh during a preprocess. We present two reordering techniques, a fast greedy strip-growing algorithm and a local optimization algorithm. The strip-growing algorithm performs lookahead simulations of the cache to adapt strip lengths to the cache capacity. The local optimization algorithm improves this initial result by exploring a set of perturbations to the face ordering. The resulting cache miss rates are comparable to the efficiency of the earlier mesh buffer scheme described by Deering and Chow, even though the vertex cache is not actively managed.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture; I.3.3 [Computer Graphics]: Picture/Image Generation - Display algorithms;

Additional Keywords: geometry compression, triangle strips.

1 INTRODUCTION

Graphics performance in low-end computer systems has recently experienced significant growth due to the integration of 3D graphics functions into custom VLSI graphics processors. The graphics subsystem now shares many similarities with the central processor. Both consist of a massively integrated processing unit, a local memory cache, and a bus to main memory (see Figure 1). Reducing the *von Neumann* bottleneck between the CPU and main memory has been a fundamental problem in computer architecture. The graphics subsystem now experiences a similar bottleneck.

In the traditional polygon-rendering pipeline, the graphics processor must access two types of information from memory: (1) the model geometry and (2) the raster images (e.g. texture map, bump map, environment map) used in shading this geometry. The problem of reducing texture image bandwidth has been studied recently [7]. In this paper, we address the problem of reducing geometry bandwidth.

The model geometry is usually described as a mesh of triangle faces sharing a common set of vertices (Figure 4a). On average, each mesh vertex is shared by 6 adjacent triangles. Vertex data, which may include position, normal, colors, and texture coordinates,

Web: http://research.microsoft.com/∼hoppe/

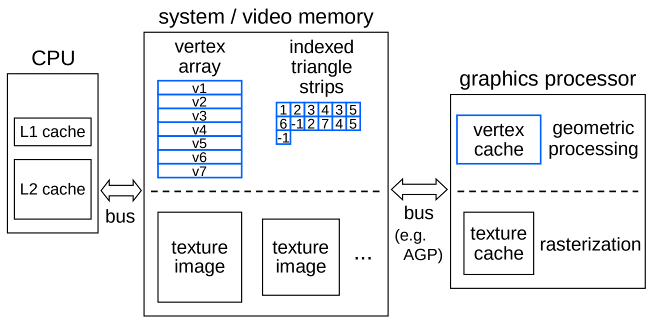


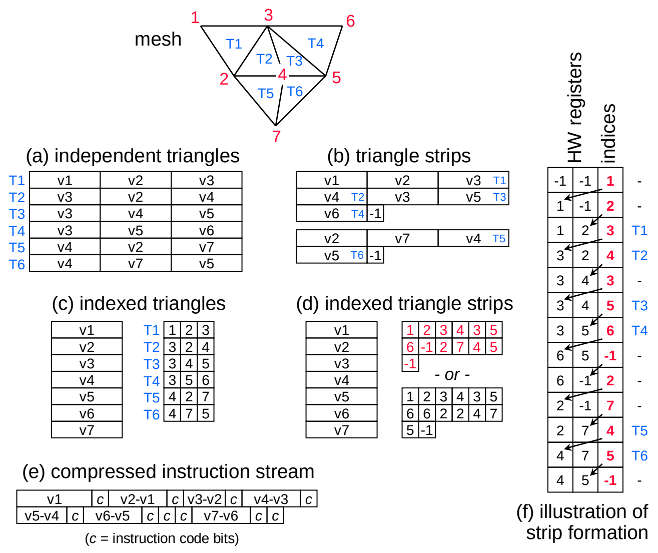
Figure 1: System architecture.

requires on the order of 32 bytes, so it is desirable to minimize the number of times this data must be read from memory. One common technique for reducing the geometry bandwidth (by a factor of almost 3) is to organize faces into triangle strips, so that two vertices are re-used between successive faces [5, 12]. Implementing such triangle strips requires a set of 3 vertex registers in the graphics processor.

The use of a larger vertex register set has the potential to further reduce geometry bandwidth by another factor of nearly 2. The key is to reorder the faces within the triangle mesh so as to maximize references to vertices already loaded in registers. Such an approach was pioneered by Michael Deering [4], and further developed by Mike Chow [3]. In their framework, the vertex data is quantized and delta-encoded into a compressed geometry stream. This geometry stream includes “push bits” to explicitly specify which vertices should be loaded into a first-in-first-out (FIFO) *mesh buffer*. Deering and Chow [3, 4] report excellent compression rates of 3–8 bytes per triangle.

In this paper, our approach is to improve locality of vertex references through a traditional application programming interface (API) for meshes. We investigate the use of a *vertex cache* in the graphics processor to *transparently* buffer data for recently referenced vertices. During a preprocess, the faces of a mesh are reordered to maximize references to vertices in the cache.

Because the traditional mesh API does not compress vertex data, the bandwidth savings in transparent vertex caching are more modest than those obtained by Deering and Chow. However, the framework offers several practical benefits. Because the vertex data is stored in native floating-point format, it can be efficiently modified from frame to frame by the application to create dynamic models. For instance, animated skinned meshes represent a significant portion of geometric bandwidth in some recent computer games [2]. Moreover, an existing application program requires no modification since it continues to use the same API. All that is necessary is to preprocess its geometric models to appropriately reorder the mesh faces. Finally, the approach provides backward compatibility since these preprocessed models still render efficiently using the same API on legacy hardware optimized for triangle strips.


mesh
1
3
6
2
4
5
7
T1
T2
T3
T4
T5
T6
(a) independent triangles
(b) triangle strips
(c) indexed triangles
(d) indexed triangle strips
- or -
(e) compressed instruction stream
(c = instruction code bits)
HW registers
indices
(f) illustration of strip formation

(a) independent triangles

T1	v1	v2	v3
T2	v3	v2	v4
T3	v3	v4	v5
T4	v3	v5	v6
T5	v4	v2	v7
T6	v4	v7	v5

(b) triangle strips

v1	v2	v3 T1	
v4 T2	v3	v5 T3	
v6 T4	-1		

v2	v7	v4 T5
v5 T6	-1	

(c) indexed triangles

v1
v2
v3
v4
v5
v6
v7

T1	1	2	3
T2	3	2	4
T3	3	4	5
T4	3	5	6
T5	4	2	7
T6	4	7	5

(d) indexed triangle strips

v1
v2
v3
v4
v5
v6
v7

1	2	3	4	3	5
6	-1	2	7	4	5
-1					

- or -

1	2	3	4	3	5
6	6	2	2	4	7
5	-1				

(e) compressed instruction stream

v1	*c*	v2-v1	*c*	v3-v2	*c*	v4-v3	*c*
v5-v4	*c*	v6-v5	*c*	*c*	*c*	v7-v6	*c* *c*

(*c* = instruction code bits)

(f) illustration of strip formation

HW registers		indices	
-1	-1	1	-
1	-1	2	-
1	2	3	T1
3	2	4	T2
3	4	3	-
3	4	5	T3
3	5	6	T4
6	5	-1	-
6	-1	2	-
2	-1	7	-
2	7	4	T5
4	7	5	T6
4	5	-1	-

Figure 2: Memory organizations for representing meshes.

Mesh organization	Memory size	Transfer size
independent triangles	$96m$	$96m$
triangle strips	$32bm$	$32bm$
indexed triangles	$\approx 22m$	$102m$
indexed triangle strips	$\approx (16 + 2b)m$	$34bm$

Table 1: Memory and transfer requirements (in bytes).

We cast face reordering as a discrete optimization problem with an explicit cost function corresponding to bus traffic. To approach this problem, we first present a greedy strip-growing algorithm for reordering the faces in a mesh to improve locality. It is inspired by the method of Chow [3]. It differs in that it explicitly simulates the behavior of the vertex cache through a lookahead procedure. The cache miss rates resulting from this algorithm are comparable to those reported by Chow, despite the fact that the mesh interface lacks explicit cache management (e.g. "push bits").

We also explore a local optimization scheme to further improve the result of the greedy strip-growing algorithm. This optimization scheme uses several operators to locally perturb the face ordering. Although the optimization scheme is several orders of magnitude slower, it is effective at further reducing vertex-cache miss rates by several percent.

2 REPRESENTATIONS FOR MESHES

In this section we briefly review various memory organizations for representing triangle meshes, and analyze the bus traffic necessary for the graphics processor to render the meshes.

Let n denote the number of vertices in the mesh, and m the number of triangle faces. Often, we use the approximation $m \approx 2n$. Vertex data is assumed to require 32 bytes (3 words for position, 3 words for normal, and 2 words for texture coordinates). Vertex data may be more compact if the normal or texture coordinates are omitted. However, to support multi-texturing, several graphics API's now allow specification of multiple texture coordinates per vertex, so vertex data may also be larger. Some of the representations refer to vertices through indices; each index is assumed to occupy 2 bytes. Although this constrains the maximum mesh size to approximately 128K faces, more complex models are commonly represented as collections of smaller, independent meshes. The mesh representations are illustrated in Figure 2, and a summary of the analysis appears in Table 1.

2.1 Traditional representations

Independent triangles The mesh is organized as an array of m faces, each containing data for its 3 face vertices, for a total of $m{\cdot}3{\cdot}32 \approx 96m$ bytes. Although this organization is seldom used in memory, many graphics drivers convert other representations into such a stream when sending the data to the graphics system.

Triangle strips The mesh faces are organized into sequences of contiguous faces called strips. The first face in the strip is specified by three vertices, and each subsequent face uses one additional vertex. Some interfaces (e.g. IRIS GL) allow explicit control over the direction of strip formation in *generalized triangle strips*. More recent, memory-based representations define *sequential triangle strips*, in which the direction of strip formation alternates left/right [12]. The default strip direction can be overriden by duplicating a vertex in the data stream, for instance vertex 3 in Figures 2b,d,f. The overall size of the representation is $32bm$ bytes, where b is a strip "bloat" factor to account for the costs of restarting strips and overriding strip direction. Typically, $1.1 \leq b \leq 1.5$. Evans et al. [5] and Xiang et al. [16] present techniques for generating good triangle strips, that is, minimizing b.

Indexed triangles The mesh is organized as an array of vertices, and an array of faces where each face refers to its 3 vertices through indices. The memory representation has size $n{\cdot}32 + m{\cdot}3{\cdot}2 \approx 22m$ bytes. Although this representation is more concise than triangle strips, the graphics processor must read more data from memory, a total of $m{\cdot}3{\cdot}(2 + 32) = 102m$ bytes.

Indexed triangle strips Again, the mesh consists of a vertex array and faces that refer to these vertices through indices, but here the faces are organized into strips. For example, such an interface is implemented in Microsoft Direct3D by the DrawIndexedPrimitiveVB(D3DPT_TRIANGLESTRIP,...) function call. We assume that a strip is restarted using a special vertex index "–1" (or alternatively by duplicating 2 indices) as shown in Figure 2d,f. Memory use is $n{\cdot}32 + m{\cdot}b{\cdot}2 \approx (16 + 2b)m$ bytes, and transfer size is $34bm$ bytes. This will be the mesh API used in the remainder of the paper.

Edge-based representations Programs commonly use more general pointer-based data structures (e.g. winged-edge, half-edge, and quad-edge) to allow traversal and topological modification on meshes. However, since many applications may find these operations unnecessary, it is preferable to use a simpler, leaner representation for the API.

2.2 Compressed instruction streams

The compression of triangle meshes has recently been an active area of research. Taubin and Rossignac [14] record trees over both the graph and the dual graph of a mesh to compress connectivity to 1–2 bits per triangle, and use a linear predictor to compress vertex data to 5–10 bytes per triangle. Gumhold and Strasser [6] present a fast scheme for encoding mesh connectivity in approximately 2 bits per triangle. Touma and Gotsman [15] encode mesh connectivity by recording the number of neighbors for each vertex, and use a "parallelogram rule" for predicting vertex positions. Hoppe [9], Li et al. [11], and Taubin et al. [13] describe compressed representations that permit progressive transmission of meshes.

While all of these schemes provide significant gains over traditional mesh representations, their decompression algorithms involve data structures that do not easily map onto a graphics processor. Therefore they are most appropriate for transmission and archival purposes. Another limitation is that these schemes currently consider only static geometry, and it would be infeasible to recompress animated geometry changing at every frame.

Bar-Yehuda and Gotsman [1] investigate the use of a vertex stack in reducing the data sent to the graphics system. They show that a stack of size $\theta(\sqrt{n})$ is both necessary and sufficient to render an arbitrary mesh without sending vertices multiple times.

Deering [4] designs a compression scheme specifically aimed at hardware implementation. The scheme makes use of a 16-entry FIFO mesh buffer. The mesh is represented as a stream of variable-length instructions that load vertices into the buffer and use buffer entries to form generalized triangle strips. Vertex data is quantized and delta-encoded to exploit coherence between neighboring vertices. Chow [3] describes several enhancements to this approach, including a *meshification* algorithm and an adaptive quantization technique. As with other compressed stream representations, the scheme is limited to static geometry.

3 TRANSPARENT VERTEX CACHING

The transparent vertex caching framework uses the indexed triangle strip memory organization described in Section 2.1. Thus, memory size requirement is still approximately $(16 + 2b)m$ bytes. However, transfer bandwidth is reduced through the introduction of a vertex cache of size k, as illustrated in Figure 1. Vertex caching reduces transfer size to $m\cdot b\cdot 2 + m\cdot r\cdot 32 = (r\cdot 32 + b\cdot 2)m$ bytes, where r denotes the average cache miss rate, in misses per triangle. Since each vertex must be loaded into the cache at least once and $m \leq 2n$, the miss rate r has a lower bound of 0.5 . The cache replacement policy is chosen to be FIFO as discussed further in Section 7.

As the approach is most closely related to the previous scheme of Deering and Chow, we review here the key differences. Recall the main characteristics of their framework:

- The graphics system reads a linear stream of vertex data and instructions. Vertex data may appear multiple times if it is re-used after being dropped from the cache.
- Vertex data is quantized and delta-encoded.
- The API is a special streaming format.
- Geometry must be static, because (1) duplicated vertices would require additional bookkeeping, (2) delta-encoding prevents random access and modification, and (3) frame-rate re-compression would be infeasible.
- Explicit bits manage allocation within the mesh buffer.

In contrast, with transparent vertex caching:

- The graphics system reads a stream of indices addressing a common array of vertices, so vertex data is not duplicated.
- Vertex data is in native uncompressed format.
- Since the API is a traditional mesh interface, applications can experience speedup without modification, and rendering is still efficient on legacy hardware.
- Geometry can be dynamic, since the application can freely modify the vertex array at video rates.
- Vertex caching is transparent and follows a strict FIFO policy.

4 FACE REORDERING PROBLEM

Maximizing the performance of the transparent vertex caching architecture gives rise to the following problem: given a mesh, find a sequence of indexed triangle strips that minimizes the amount of data transferred over the bus. The sequence of triangle strips is uniquely defined by a permutation F of the original sequence of faces $\hat{F}$. Thus, the general optimization problem is

$$\min_{F\in\mathcal{P}(\hat{F})} \mathcal{C}(F)$$

where $\mathcal{P}(\hat{F})$ denotes all $m!$ permutations of the faces, and the cost

$$\mathcal{C}(F) \;=\; m\left(r(F)\cdot 32 + b(F)\cdot 2\right) \qquad (1)$$

corresponds to the number of bytes transferred over the bus. The hardware model is that, for each face, the graphics processor requests 3 vertices from the cache, in the order shown in Figure 2f.

For example, Figure 4 shows the costs for three different orderings of the faces in a simple mesh. The ordering is illustrated using the black line segments (for adjacent faces within a strip) and white line segments (for strip restarts). Within each face, the colors at the three corners indicate if the vertex was present in the cache of size $k = 16$. As shown in Figure 4b, stripification algorithms may produce strips that are too long, resulting in a cache miss rate of $r \approx 1.0$, observed visually as one red corner per triangle. In contrast, our reordering techniques (Figures 4c-d) come closer to the optimal $r = 0.5$, i.e. one cache miss every other triangle.

Since $r \approx 0.6$ and $b \approx 1.5$, the vertex cache miss traffic $(r\cdot 32)$ is generally much more significant than the vertex index traffic $(b\cdot 2)$. Both our face reordering algorithms make some simplifying approximations with respect to this second, less significant term.

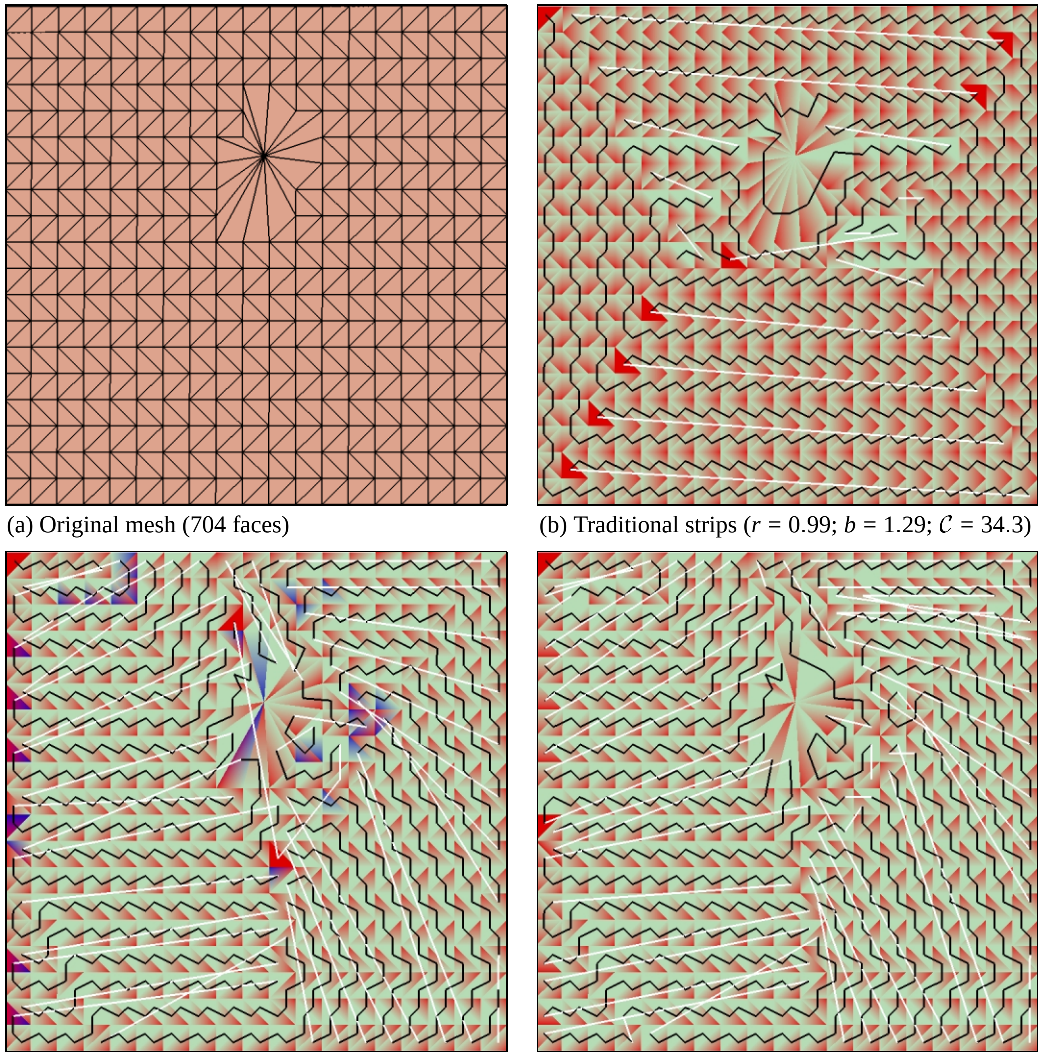
```
function greedy_reorder()
Sequence<Face> F={};       // new face sequence
Face f=0;
loop
    if (!f)                // restart process at some location
        f=some_unvisited_face_with_few_unvisited_neighbors();
        if (!f) break;     // all faces are visited
    Queue<Face> Q;         // possible locations for strip restarts
    loop                   // form a strip
        if (strip_too_long())  // using lookahead simulation
            f=Q.next_unvisited_face();  // may be 0
            break;         // force a strip restart
        f.mark_visited()
        F.add_to_end(f);
        // Get counter-clockw. and clockwise faces continuing strip
        (fccw,fclw) = f.next_two_adjacent_unvisited_faces();
        if (fccw)          // continue strip counter-clockwise
            if (fclw) Q.push(fclw);
            f=fccw;
        else if (fclw)     // continue strip clockwise
            f=fclw;
        else               // cannot continue strip
            f=Q.next_unvisited_face();  // may be 0
            break;         // force a strip restart
return F;
```

Figure 3: Pseudocode for the greedy strip-growing algorithm.

5 GREEDY STRIP-GROWING TECHNIQUE

Our first approach to solving the face reordering problem is a simple greedy technique. It is fast and can be used to seed the latter local optimization technique with a good initial state. The basic strategy is to incrementally grow a triangle strip, and to decide at each step whether it is better to add the next face to the strip or to restart the strip. This binary decision is made by performing a set of lookahead simulations of the vertex-cache behavior. Pseudocode for the algorithm in shown in Figure 3; we next present it in more detail. The output of the algorithm is shown in Figure 4c.

The algorithm begins by marking all faces of the mesh as unvisited. The first visited face is chosen to be one with the fewest number of neighbors. From this face, the algorithm begins growing a strip. If there are two neighboring unvisited faces, it always continues the strip in a counter-clockwise direction, but pushes the



(a) Original mesh (704 faces)

(b) Traditional strips (r = 0.99; b = 1.29; $\mathcal{C}$ = 34.3)

(c) Greedy strip-growing (r = 0.62; b = 1.28; $\mathcal{C}$ = 22.3)

(d) Local optimization (r = 0.60; b = 1.32; $\mathcal{C}$ = 21.7)

Figure 4: A comparison of the face orderings resulting from (b) a traditional stripification algorithm, (c) the greedy strip-growing algorithm, and (d) the local optimization algorithm. The result of simulating a 16-entry FIFO vertex cache is shown using the corner colors (green=cache hit; red=cache miss; blue=cache miss in (c) eliminated in (d)). Indicated results are: the average number r of cache misses per triangle, the strip bloat factor b, and the overall bandwidth cost $\mathcal{C}$ in bytes per triangle.

other neighboring face onto a queue Q of possible locations for strip restarts. If there are no neighboring unvisited faces, it cannot continue the strip and therefore restarts a new strip at the first unvisited face in Q and then clears Q. If there are no unvisited face in Q (i.e. the algorithm has painted itself into a corner), the process must be restarted at a new location on the mesh. In selecting this new location, the primary criterion is to favor unvisited faces with vertices already in the cache. A secondary criterion is to select a face with the fewest number of unvisited neighbors.

Because the algorithm described so far does not constrain the lengths of strips, the strips could overflow the capacity of the cache, thereby preventing re-use of vertices between successive strips. Therefore, before adding each face, the algorithm performs a lookahead simulation to decide if it should instead force the strip to restart. Specifically, it performs a set of s simulations $\{0 \ldots s{-}1\}$ of the strip-growing process over the next s faces. (The choice of s will be given shortly.) Simulation number $i \in \{0 \ldots s-1\}$ forces the strip to restart after exactly i faces, and computes an associated cost value $C(i)$ equal to the average number of cache misses per visited face. If among these simulations, the lowest cost value corresponds to restarting the strip immediately, i.e.

$$\forall i \in \{1 \ldots s-1\} \quad C(0) < C(i) \ ,$$

the strip is forced to restart. Through experimentation, we have found $s = k + 5$ to be a good choice for a FIFO cache.

Note that the local cost function C approximates only the first term of the true bandwidth cost $\mathcal{C}$ of Equation 1. Although C fails to account for vertex index traffic, the greedy algorithm does implicitly attempt to minimize the number of strips, since restarts are only allowed when all the strict inequalities above are satisfied. Within each strip, the algorithm cannot afford to leave isolated faces behind, so it has little choice over the direction of strip formation.

As an optimization, instead of computing all s cost values $\{C(0) \ldots C(s{-}1)\}$ before visiting each face, the algorithm first computes $C(0)$ and then stops as soon as it finds another $C(i) \leq C(0)$. Also, the first cost value computed after $C(0)$ is $C(i_{min})$ where i_{min} was the simulation returning the lowest cost value for the previously visited face. With this optimization, the number of lookahead simulations per face is reduced from $k + 5 = 21$ to 2.9 on average.

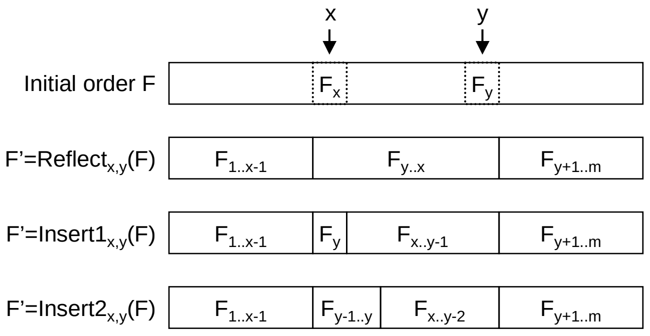


Figure 5: Perturbations to the face ordering.

6 LOCAL OPTIMIZATION TECHNIQUE

In this second technique, we start with the initial sequence of faces F produced by the greedy algorithm, and attempt to improve it through a set of ordering perturbations. For each candidate perturbation $P : F \to F'$, we compute the change in cost $\Delta C(P) = C(F') - C(F)$ and apply the perturbation if $\Delta C(P) < 0$. In the next sections we describe the cost $C(F)$ approximating the true cost $\mathcal{C}$ from Equation 1, the types of reordering perturbations $P : F \to F'$, the process of selecting candidate perturbations, and several techniques that improve efficiency and quality of results.

Cost metric The primary cost function is

$$C(F) \;=\; 32 \cdot m \cdot r_k(F) \;+\; 6 \cdot \#\text{strips}(F)\;,$$

where $m \cdot r_k(F)$ denotes the total number of cache misses for a cache of size k, and #strips(F) is the number of triangle strips induced by the face sequence F. This cost function is an approximation of the true cost function $\mathcal{C}$ from Equation 1 in that it does not measure the number of duplicated vertices necessary to override the default direction for strip formation. In our opinion, this difference does not significantly affect results.

Reordering perturbations As shown in Figure 5, we define three types of perturbation (subsequence reflection, insertion of one face, and insertion of two faces), each parametrized by two indices $1 \le x, y \le m$ into the sequence F. We chose these three types of perturbation because they require only two parameters and yet have enough freedom to find many reordering improvements. Let $P^t_{x,y}$ denote a perturbation of type t.

Selection of candidate perturbations Recall that each candidate perturbation $P^t_{x,y}$ is parametrized by two face indices x and y. To determine the index x, we simply visit all the faces in a random order. For each visited face f, we find its index x in the current ordering, i.e. $F_x = f$.

Having selected x, we form a set Y of indices of possible parameters y. We could let Y be the exhaustive set $\{1 \ldots m\}$, but that would be wasteful since most faces F_y would be nowhere near F_x and thus unlikely to contribute to an improvement. We therefore let Y contain the indices of faces either vertex-adjacent to F_x in the mesh or adjacent to F_x in the current ordering (i.e. F_{x-1} and F_{x+1}).

For each $y \in Y$, we attempt all three types of perturbation, and find the one returning the lowest cost:

$$\min_{y,t} C(P^t_{x,y}(F))\;.$$

If $\Delta C(P^t_{x,y}) \ge 0$, we are unable to find a beneficial operation, and therefore proceed to the next x. Otherwise, $P^t_{x,y}$ is beneficial and could be applied at this point.

However, before committing $P^t_{x,y}$, we first see if we can find a locally better perturbation. Specifically, we keep the index y and determine the other index

$$z = \operatorname*{argmin}_{z' \in Z} \min_t C(P^t_{y,z'}(F))$$

with the best perturbation from y, where the set Z is formed like Y. If $z = x$ then we have found a locally optimal perturbation, and we apply it. Otherwise, we replace $x \leftarrow y$ and $y \leftarrow z$, and iterate again until convergence.

Fast cost re-evaluation For reasonable performance, the computation of $\Delta C(P^t_{x,y})$ should be fast and independent of the interval length $|x - y|$. Let us first consider just the two perturbations Insert1$_{x,y}$ and Insert2$_{x,y}$. One key observation is that the cache behavior for the sequences F and F' is likely to be different only near the interval endpoints x and y, since the cache generally resynchronizes within the interior of the interval if x and y are far apart. To exploit this, our approach is as follows.

For each face F_i we store a set b_i of three bits equal to the current cache-miss states of its three vertex references. Given the perturbation $P^t_{x,y} : F \to F'$, we first load up the expected cache state just prior to location x by moving backwards through F from x until k misses have been detected in the stored bits b_i. Next, we simulate the cache from x forwards through F', recording changes in cache misses from those stored in the b_i. When k successive cache misses are detected without any intervening cache-miss changes between F and F', the cache state is known to be resynchronized, and thus no more changes will occur until y is reached. Note that the number of faces visited before the caches resynchronize is generally independent of the interval size $|x - y|$.

We then perform the same procedure for the sequence beginning at y. Finally, the last element necessary to compute $\Delta C(P)$ is to determine the induced change in the number of triangle strips. For this, we need only consider the face adjacencies at the slice points used by P (shown in Figure 5).

The Reflect$_{x,y}$ perturbation is more difficult to handle because the entire interval $F_{x\ldots y}$ is reversed. For fast evaluation of its change in cost, we store at each face F_i another three bits b^R_i corresponding to the cache-miss states when traversing the faces of F in *reverse order*, and use those when simulating $F_{y..x} \subset F'$.

Secondary cost function Because the cost function C is rather flat and the perturbations do not look very far, we smooth out the cost function by adding a secondary cost function

$$C'(F) \;=\; 0.003\, m \cdot r_{k-1}(F) \;+\; 0.002\, m \cdot r_{k+1}(F)$$

that examines the number of cache misses for caches with one less entry ($r_{k-1}(F)$) and with one more entry ($r_{k+1}(F)$). The motivation for this function is that it attempts to maximize unused space in the cache whenever possible, in order to preserve "slack" for possible future improvements.

Search pruning It is unlikely that a perturbation will be beneficial if its endpoint x lies in the middle of a strip and the surrounding faces have good caching behavior. Therefore, we use the simple heuristic of pruning the search from x if the face F_x is neither at the beginning nor at the end of a strip and the sum of cache misses on the three faces $\{F_{x-1}, F_x, F_{x+1}\}$ is less than 3.

7 RESULTS

Cache replacement policy Our very first experiments involved a vertex cache with a least-recently-used (LRU) replacement policy, since this policy usually performs well in other contexts. However, as shown in Figure 6, we soon found that an LRU cache cannot support strips as long as a FIFO cache. The reason is that vertices shared between a strip $s-1$ and the next strip s are referenced during the traversal of s, and thus "pushed to the front" of the

LRU cache, even though they are no longer used in the subsequent strip $s+1$. (For example, see vertices "2" and "3" in Figure 7.) In contrast, when a FIFO cache reaches steady state, vertices between strips $s-1$ and s are dropped from the cache at precisely the right time — before any vertices used for strip $s+1$ (Figure 7). On a regular mesh, the optimal strip length appears to be only $k-2$ faces for an LRU cache versus $2k-4$ faces for a FIFO cache. We therefore adopted the FIFO policy.

Cache size Figure 8 plots cache miss rate r as a function of cache size k using different runs of the greedy strip-growing algorithm. Reordering algorithms depend strongly on the parameter k. A mesh preprocessed for a cache of size k will be sub-optimal on hardware with a larger cache, and more importantly, it may completely thrash a smaller cache. For most of our examples, simulating the face orderings optimized for $k = 16$ on a cache of size $k = 15$ increases the average cache miss rate r by 10–30%.

Surprisingly, it is theoretically feasible for an element sequence to perform better on a FIFO cache of size k than on a FIFO cache of size $k+1$.[1] For instance, in the limit the sequence

$$6, 1, 7, 2, 4, 6, (1, 2, 3, 4, 5, 6, 7), (1, 2, 3, 4, 5, 6, 7), \ldots$$

has twice as many misses on a cache of size 5 than on one of size 4. It might therefore be possible that a super-optimized face ordering would have a similar performance dependency on the precise cache size. However, we have never seen this behavior in practice.

Greedy strip-growing The columns labeled "Greedy" in Table 2 show results of the greedy strip-growing algorithm of Section 5. Two of the results are pictured in the first column of Figure 9.

For the mesh buffer scheme [3, 4], Chow reports buffer load rates ranging from 0.62 to 0.70 vertices per triangle. Two of these meshes are the same ours; he reports 0.62 for the "bunny" and 0.70 for the "schooner".[2] The cache miss rates for our greedy algorithm are therefore comparable to those results, even though the cache is not actively managed.

The "gameguy" mesh has a notably high miss rate. It is due to the fact many of its vertices have multiple normals and are therefore artificially replicated in the mesh, resulting in many boundary edges and a low face-to-vertex ratio. Such duplication of vertices also occurs in the "fandisk" and "schooner" meshes. A performance number that better accounts for this variability in m/n is the average number of times each mesh vertex is loaded into the cache (labeled "miss/vertex" in Table 2), which has a lower bound of 1.

The three meshes with the highest miss/vertex ratios are "bunny2000", "bunny4000" and "buddha". These meshes are precisely the ones obtained as the result of mesh simplification. The high miss rates are probably due to the irregular mesh connectivities resulting from the geometrically optimized simplification process.

The execution rate of the greedy algorithm on all of these models ranges from 35,000 to 43,000 faces/second on a 450 MHz Pentium 2 system. So even the most complex mesh is processed in under 6 seconds.

Local optimization Results of the local optimization algorithm of Section 6 are presented in the columns labeled "Optimiz." in Table 2 and in Figures 9b,d.

The results show that local optimization is generally able to reduce cache misses by 3–6%. This gain is somewhat disappointing, as we were hoping to see greater improvements. The results seem to indicate, however, that the solution of the greedy algorithm is near a local minimum of the bandwidth cost $\mathcal{C}$.

[1]Thanks to John Miller for pointing this out.

[2]Chow also reports 0.63 for a buddha model of 293,233 faces but it is unfortunately not the same mesh as our simplified 100,000-face buddha.

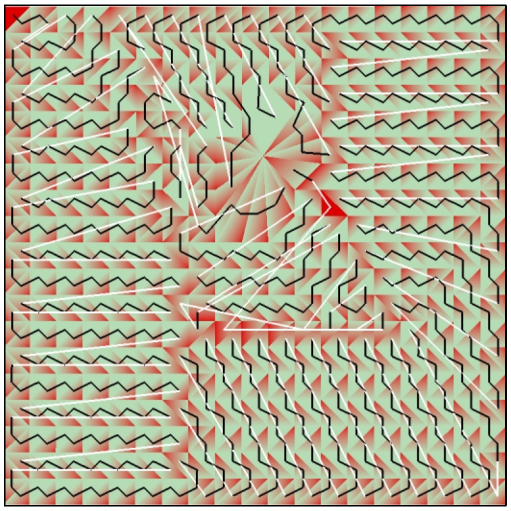
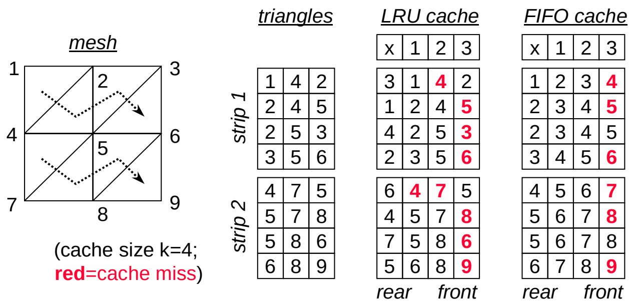


Figure 6: Output of the greedy strip-growing algorithm using an LRU cache replacement policy instead of FIFO, with cache size $k = 16$ ($r = 0.66$; $b = 1.40$; $\mathcal{C} = 23.9$). Compare with Figure 4(c).



	triangles			LRU cache				FIFO cache			
				x	1	2	3	x	1	2	3
strip 1	1	4	2	3	1	**4**	2	1	2	3	**4**
	2	4	5	1	2	4	**5**	2	3	4	**5**
	2	5	3	4	2	5	**3**	2	3	4	5
	3	5	6	2	3	5	**6**	3	4	5	**6**
strip 2	4	7	5	6	**4**	**7**	5	4	5	6	**7**
	5	7	8	4	5	7	**8**	5	6	7	**8**
	5	8	6	7	5	8	**6**	5	6	7	8
	6	8	9	5	6	8	**9**	6	7	8	**9**

Figure 7: Comparison of LRU and FIFO on a simple mesh.

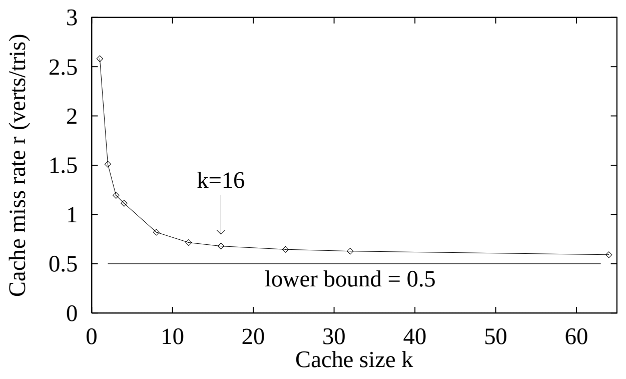


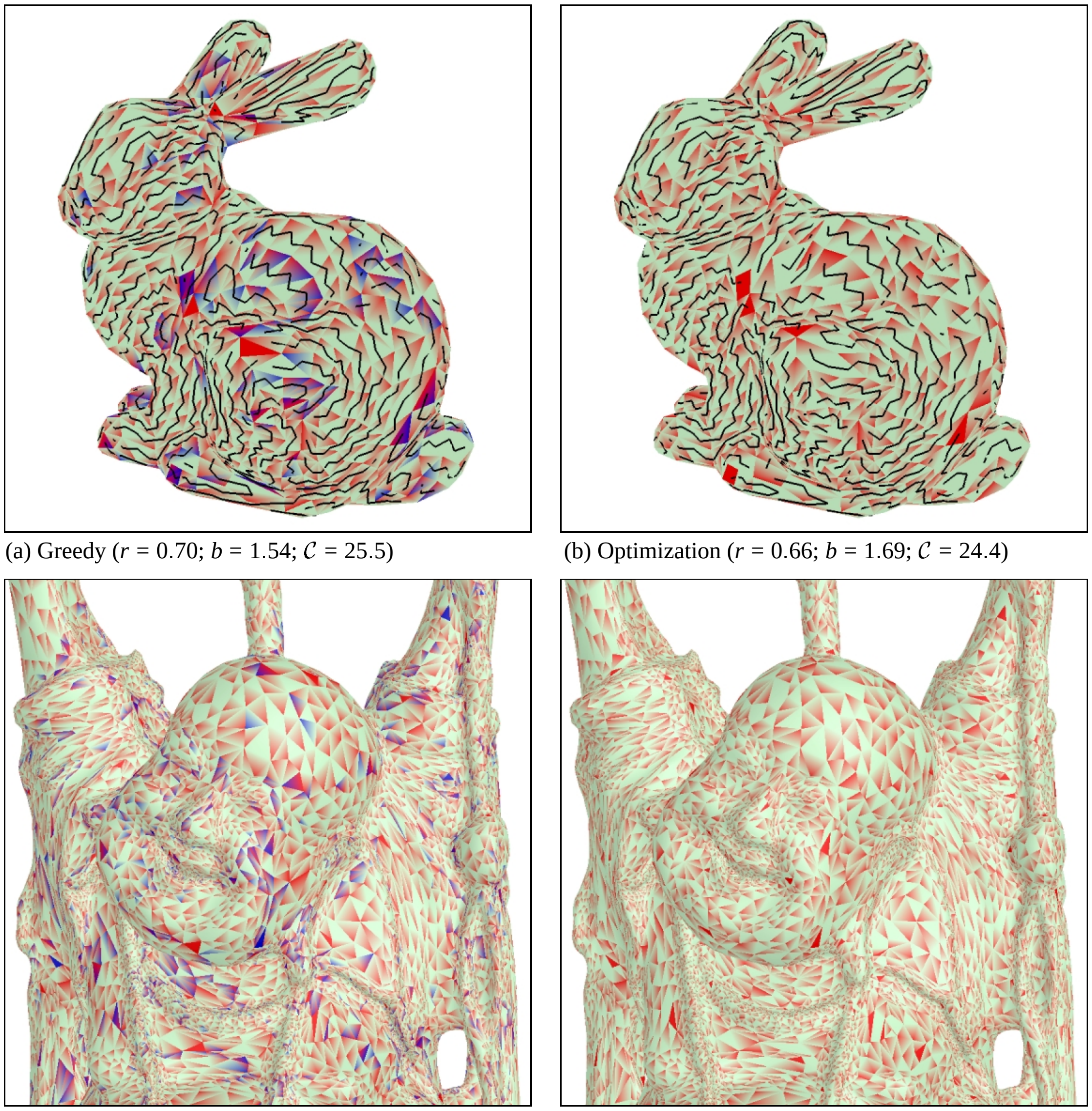
Figure 8: FIFO cache effectiveness as a function of cache size with the greedy strip-growing algorithm on the 4000-face bunny.

Also, one must keep in mind that the cache miss rate has an absolute lower bound of 1 miss per vertex since each vertex must be loaded at least once into the cache. For most meshes, the lower bound is in fact higher because the maximum lengths of strips is bounded by the cache size, and non-boundary vertices on the ends of strips must loaded in the cache more than once. For an infinitely large regular triangulation, the number of misses per vertex therefore has a lower bound of $1 + \frac{1}{k-1}$.

Execution times for the algorithm range from 5 minutes to 4 hours on these meshes. The algorithm finds improvements at a high rate initially, then gets diminishing returns, so it could be stopped earlier.

Data set	#vertices	#faces	Average cache miss rate				Bandwidth (bytes/tri)		
	n	m	r = miss/triangle		miss/vertex		Triangle	Vertex caching	
			Greedy	Optimiz.	Greedy	Optimiz.	strips	Greedy	Optimiz.
grid20	391	704	0.62	0.60	1.11	1.07	36.7	22.3	21.7
grid40	1,075	1,999	0.67	0.63	1.25	1.17	43.9	24.7	23.6
fandisk	7,233	12,946	0.61	0.60	1.09	1.08	37.4	22.3	22.1
gameguy	7,874	10,000	0.88	0.86	1.12	1.09	46.9	31.6	30.8
bunny2000	1,015	1,999	0.70	0.66	1.38	1.30	45.0	25.5	24.4
bunny4000	2,026	3,999	0.68	0.65	1.34	1.27	44.2	24.8	23.8
bunny	34,835	69,473	0.64	0.62	1.28	1.24	39.8	23.4	22.8
buddha	49,794	100,000	0.70	0.65	1.40	1.30	45.8	25.5	24.2
schooner	105,816	205,138	0.62	0.61	1.20	1.19	41.2	22.8	22.6

Table 2: Cache miss rates using the greedy strip-growing algorithm and the local optimization algorithm (expressed as both miss/triangle and miss/vertex), and overall transfer bandwidth using traditional triangle strips versus transparent vertex caching.



(a) Greedy (r = 0.70; b = 1.54; $\mathcal{C}$ = 25.5)

(b) Optimization (r = 0.66; b = 1.69; $\mathcal{C}$ = 24.4)

(c) Greedy (r = 0.70; b = 1.63; $\mathcal{C}$ = 25.5)

(d) Optimization (r = 0.65; b = 1.70; $\mathcal{C}$ = 24.2)

Figure 9: Results of greedy strip-growing (left column) and local optimization (right column) with a 16-entry FIFO vertex cache, for bunny2000 and buddha. (Blue corners on the left indicate cache misses eliminated on the right.) Captions refer to the average number r of cache misses per triangle, the strip bloat factor b, and the overall bandwidth cost $\mathcal{C}$ in bytes per triangle.

Analysis The rightmost columns of Table 2 compare the total bandwidth requirements for a traditional triangle strip representation and for the transparent vertex caching framework. It demonstrates that bandwidth is reduced by a factor of approximately 1.6 to 1.9 .

8 DISCUSSION

Issues in modifying rendering order Modifying the order in which faces are rendered may alter the final image if faces are coincident, if the Z-buffer is disabled, or if the triangles are partially transparent. This limitation is shared by all schemes that modify the face ordering, including ordinary triangle strip generation.

Vertex data compression With the transparent vertex caching framework, vertex data can be compressed by the CPU independently of mesh connectivity. In particular, time-dependent geometry presents a significant opportunity for vertex data compression. As an example, Lengyel [10] describes a scheme that clusters vertices together and predicts their positions by associating to each cluster a local coordinate frame that deforms over time; the resulting residuals are compressed separately. In effect, Lengyel's scheme reorders vertices to improve geometric coherence, and does not concern itself with the order of faces. On the other hand, our framework reorders faces to improve graphics coherence, and does not care about the order of vertices. This demonstrates how vertex data compression could interact elegantly with our framework.

Memory access pattern for vertex data As the results in Table 2 indicate, a large percentage of vertices are loaded into the cache only once, i.e. the first and only time they cause a cache miss. In some system architectures, it may be useful to reorder the vertices in the mesh to match the order in which they are first requested, so that the memory access pattern is mostly sequential. The trade-off is that reordering the vertices causes some loss of transparency, since the application may need to be aware that the mesh vertices have been permuted.

In our opinion, the memory access pattern is not a stumbling block. Unlike in a general CPU computation, the memory access pattern from the graphics processor can be predicted by buffering the vertex index stream (which is entirely sequential), so memory latency becomes less important than overall memory bandwidth. Several graphics systems already perform similar buffering when pre-fetching texture memory as triangle fragments make their way to the rasterizer.

9 SUMMARY AND FUTURE WORK

We have explored the use of a vertex cache to transparently reduce the geometry bandwidth between the graphics processor and memory in the context of a traditional mesh rendering API. In many cases, it is unnecessary for the application program to be aware of this caching scheme, even if the program applies runtime deformations to the mesh geometry. Maximizing the efficiency of the cache simply involves reordering the faces in the mesh during a preprocessing step.

We have presented a greedy strip-growing algorithm for reordering the faces, and shown that, even without explicit cache management, it is able to achieve comparable results to the previous scheme by Deering and Chow. The greedy algorithm operates at an approximate rate of 40,000 faces/sec and is thus highly practical.

We have also explored a perturbation-based optimization scheme for further improving the face ordering. Although costly in terms of computation time, it succeeds in reducing bandwidth by several percent.

This project suggests a number of areas for future work:

- Exploring more complex reordering perturbations that exploit the strip structure present in the face ordering.
- Examining the interaction with texture caching. Although optimal face orderings for vertex caching and texture caching are likely different, a compromise would be feasible.
- Maintaining cache-efficient face ordering during level-of-detail (LOD) control. While this is straightforward for a precomputed set of LOD meshes, it seems difficult for continuous LOD and particularly for view-dependent LOD [8].

ACKNOWLEDGMENTS

I am extremely grateful to John Miller. Thanks also to Paolo Sabella and David Kirk of NVIDIA and to Anuj Gosalia of the Microsoft Direct3D group for helpful feedback. The bunny and buddha meshes are courtesy of the Stanford University Computer Graphics Laboratory.

REFERENCES

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

[2] Birdwell, K. Valve Corp. Personal communication, 1998.

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

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

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

[6] Gumhold, S., and Strasser, W. Real time compression of triangle mesh connectivity. *Computer Graphics (SIGGRAPH '98 Proceedings)* (1998), 133–140.

[7] Hakura, Z., and Gupta, A. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th International Symposium on Computer Architecture* (June 1997), pp. 108–120.

[8] Hoppe, H. View-dependent refinement of progressive meshes. *Computer Graphics (SIGGRAPH '97 Proceedings)* (1997), 189–198.

[9] Hoppe, H. Efficient implementation of progressive meshes. *Computers and Graphics 22*, 1 (1998), 27–36.

[10] Lengyel, J. Compression of time-dependent geometry. In *Symposium on Interactive 3D Graphics* (1999), ACM, pp. 89–96.

[11] Li, J., Li, J., and Kuo, C. C. Progressive compression of 3D graphics models. In *Multimedia Computing and Systems* (April 1997), IEEE, pp. 135–142.

[12] Neider, J., Davis, T., and Woo, M. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[13] Taubin, G., Guéziec, A., Horn, W., and Lazarus, F. Progressive forest split compression. *Computer Graphics (SIGGRAPH '98 Proceedings)* (1998), 123–132.

[14] Taubin, G., and Rossignac, J. Geometric compression through topological surgery. *ACM Transactions on Graphics 17*, 2 (April 1998), 84–115.

[15] Touma, C., and Gotsman, C. Triangle mesh compression. In *Proceedings of Graphics Interface '98* (1998), pp. 26–34.

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