Geometry Engine Optimization: Cache Friendly Compressed Representation of Geometry

Jatin Chhugani
Intel Corporation,
Santa Clara, CA

Subodh Kumar*
Indian Institute of Technology,
New Delhi, India

Abstract

Recent advances in graphics architecture focus on improving texture performance and pixel processing. These have paralleled advances in rich pixel shading algorithms for realistic images. However, applications that require significantly more geometry processing than pixel processing suffer due to limited resource being devoted to the geometry processing part of the graphics pipeline. We present an algorithm to improve the effective geometry processing performance without adding significant hardware. This algorithm computes a representation for geometry that reduces the bandwidth required to transmit it to the graphics subsystem. It also reduces the total geometry processing requirement by increasing the effectiveness of the vertex cache. A goal of this algorithm is to keep the primitive assembly simple for easy hardware implementation.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; E.4.0 [Data]: Coding and Information Theory—Data Compaction and Compression.

Keywords: Graphics hardware cache, data compression, interactive navigation.

1 Introduction

In recent years, graphics hardware design has made spectacular advances in efficiency as well as in generality and programmability. These improvements have come in the form of faster processors, more memory and caches, and faster buses. On the other hand, models used in applications have also grown larger and shading more complex. They have more triangles and vertices, more attributes per vertex and longer shaders. Detailed models of airplanes, high resolution medical data, and even 3D game environments [IdSoftware 2005] consist of several million polygons and more. Indeed, we expect the size and complexity of models used in the toughest applications to remain a step ahead of the hardware. These applications will increasingly demand more computation, memory, and bandwidth.

Graphics hardware is designed for efficient rendering of triangles. The rendering pipeline starts when a user specifies the geometry and continues until pixel data is written into the frame buffer. We broadly partition rendering into three stages: data gathering, geometry operations, and pixel operations, each of which must remain separately efficient. This implies that in addition to fast vertex and fragment processing, effective interfaces to specify geometry must exist so that data gathering, which includes transmission from the application to the geometry engine, may be fast.

Data gathering and geometry processing are the subject of this paper. These may be further subdivided into per-vertex operations and per primitive operations. We focus our attention on the primitive aspects: in particular, on reducing the size of primitive representation and on reducing the computation required for each primitive. Vertex attribute compression and per vertex computation, which are surely at least as important, are complimentary to this effort and out of the scope of this paper.

Our input domain is a triangular mesh: a sequence of triangles that share vertices. Typically there are only about half as many unique vertices in a mesh as there are triangles but there are thrice as many references. A naive geometry engine would process each vertex six times, repeatedly fetching and shading it. A more efficient strip-like representation still results in each vertex being processed at least twice. For vertex shader limited applications this could halve overall performance. That is why popular graphics architectures build in a post-shader vertex cache. If this cache were large enough to fit all attributes of every vertex in the model, one would never have to repeat the processing of any vertex. However, a fast cache that large is simply not practical. That is why eight or sixteen entry caches are commonly used. If vertex-references in a list of triangles are near each other, the cache is well utilized. We devise a compressed primitive-topology representation that ensures high vertex coherence. Furthermore, in addition to a high compression ratio, we ensure the simplicity of the decompression algorithm, so its overhead is negligible.

We have taken the minimalist approach of primarily staying within the constraints of the currently popular architecture. We do require that the triangles be specified in an order suitable for efficient processing. Sometimes applications need to maintain total control over it. However, most applications do not care about the specific order of primitives that share a common state, especially if the order is repeatable for a static model.

For a model with n unique vertices, each of which must be processed at least once, a minimum of n cache misses are necessary. We define the *cache miss ratio* as the number of vertices per triangle that must be shaded because they are not already in cache. Thus for a model with m triangles and n vertices, the minimum ratio is $M = \frac{n}{m}$. In practice, $m \sim 2n$, and hence M is at least 0.5. In order to guarantee a cache miss ratio of 0.5, it is necessary to use a cache of size $O(\sqrt{n})$ in general [Bogomjakov and Gotsman 2001]. In reality, an unlimited cache is not practical. Although other variants are possible, for simplicity we assume a small post vertex-shading single-level FIFO cache [Nvidia 2001].

Significant vertex attribute compression is indeed possible [Purnomo et al. 2005]. However, vertex compression alone cannot always relieve the bandwidth bottleneck and mesh topology must be compressed as well. Hence it is imperative to not only make the topological information cache friendly, but to also compress it. Most research into topology compression does not focus on efficient hardware decompression and cache coherence, and *vice-versa*. That is the contribution of this paper: it compresses well,

*most of the work was done while the author was at the Johns Hopkins University, Baltimore, MD.

decompresses efficiently and maintains cache coherence, all at the same time. Although, we do not discuss the problem of vertex compression here, we note that our algorithm clusters nearby vertices together, thereby lending itself naturally to quantization based vertex attribute compression schemes like [Purnomo et al. 2005].

1.1 Previous Work

Geometry compression includes compressing vertex attributes (like position, normals, texture coordinates, etc.) along with the topology information. Vertex attribute compression algorithms [Chow 1997; Purnomo et al. 2005] are typically lossy in nature and quantize the data by exploiting the coherence between neighboring vertices. Topology (or connectivity) compression is generally lossless and has been an active area of research [Taubin and Rossignac 1998; Gumhold and Strasser 1998; Szymczak et al. 2001; Isenburg 2001; Alliez and Desbrun 2001; Kronrod and Gotsman 2002]. However, these algorithms [Szymczak et al. 2001] do not produce a cache coherent representation and typically encounter *one cache miss per triangle* for typical hardware cache sizes (8-64). Moreover, only a few algorithms are intended [Deering 1995; Chow 1997] for hardware decompression. Reducing the cache misses is of critical importance since it reduces both the time and hardware resources to fetch the pre-transformed vertex attributes when they are required during rasterization. The problem of reducing the cache misses has also been investigated separately [Bar-Yehuda and Gotsman 1996; Bogomjakov and Gotsman 2001; Hoppe 1999; Yoon et al. 2005].

Although compression algorithms suited for "polygon soups" exist [Gueziec et al. 1999; Gandoin and Devillers 2002], the most compression is achieved when models are (orientable) manifolds, or may be decomposed into manifolds [Touma and Gotsman 1998; Szymczak et al. 2001; Gumhold and Strasser 1998]. Most recent geometry compression algorithms are based on a spanning tree of the adjacency graph [Rossignac 1999] or on a separation of the vertices [Touma and Gotsman 1998]. Both classes of methods seem unsuitable for hardware decompression. Not only are the steps somewhat complex, the temporary memory required can be large. The method proposed earlier by [Deering 1995] is designed for hardware. They present a "generalized mesh" representation, which can be efficiently compressed and decompressed using a fixed memory size. This technique still requires variable length components and has not been adopted in popular architectures. Our method is inspired by this technique. We first extend it to improve its performance, and then show how the extensions can be implemented in simpler operations, which require simpler hardware. [Chow 1997] shows how to compute the generalized mesh representation from a vertex array. Triangle strips are also potentially well suited for compression [Isenburg 2001]. Generation of long triangle strips from a given vertex array has been active area of research [Estkowski et al. 2002; Xiang et al. 1999; Evans et al. 1996; Diaz-Gutierrez et al. 2006].

[Hoppe 1999] solve the problem of effective utilization of a fixed size vertex cache. The algorithm computes an order for issuing vertices by greedily growing strips of lengths appropriate for a given cache size. It further improves the strips with a local optimization technique that perturbs the triangle order to improve cache effectiveness. It also discusses size trade offs between using strips and vertex arrays. Later [Bogomjakov and Gotsman 2001] considers the theoretical problem of determining the minimal cache size that guarantees only n cache misses for a manifold with n vertices. [Yoon et al. 2005] present an out-of-core algorithm for cache oblivious layouts of meshes. It transforms the given mesh into a graph and gives an approximate solution for the permutation of the nodes which reduces the length of their edge dependency. This leads to a reduction in number of cache misses, which improves with the increase in the size of the cache. [Yoon and Lindstorm 2006] provides metrics to access the quality of meshes for block-based caches and can be used to improve the layouts for meshes. [Lin and Yu 2006] also proposes a vertex caching scheme where it formulates the problem as a linear optimization function of the mesh topology and the vertex state in a simulated cache buffer. The algorithm also handles progressive meshes by adaptively updating the reordered sequence at full detail.

1.2 Overview

We have chosen to further develop the compression technique of [Deering 1995] and [Chow 1997] to

1. incorporate cache coherence.
2. improve the compression of the geometry stream.
3. make it more suitable for current hardware.

[Deering 1995] describes a generalized triangle mesh representation, which specifies one vertex per most triangle in addition to codes for labels *Restart*, *Replace-Oldest*, *Replace-Middle* and *Push*, which specify how to connect that vertex to the latest triangle. This is a generalization of triangle strips, in which the connection of the new vertex to the latest triangle is predetermined. It also has the facility to directly refer to a "mesh buffer" address, which is akin to a vertex cache. [Chow 1997] provides an algorithm to compute a generalized mesh from a vertex array, but it is designed to maximize the reuse of cache addresses. We improve that algorithm to increase the reuse of cache entries (and thus improve coherence as well as compression). We are motivated in this algorithm by the "cache simulation" idea of [Hoppe 1999], but *lower* the cache miss ratio further. In addition, we encode even the cache-misses efficiently. Finally, in order to make the algorithm better fit the current graphics architecture, we propose conversion of the resulting generalized triangle mesh into a compact, per triangle representation.

Our experiments as well as those by others [Hoppe 1999] show that cache coherence does not significantly improve with cache replacement policies more sophisticated than FIFO. Even if we allow a geometry stream to fully control the entries that may be replaced in cache, the miss ratio M does not improve significantly. Given the simplicity of the FIFO policy, we base the rest of our design on a FIFO cache.

The main steps of our algorithm are as follows:

1. Decompose the mesh into parallel chains of vertices.
2. Induce cuts along each chain to derive the vertex issue order for the cache.
3. Generate a mesh representation that issues the vertices in the desired order.
4. Order vertices in the vertex array in the order of their first issue.
5. Entropy-encode the resulting mesh.

2 Cache Performance

In order to improve the vertex cache performance, we need to minimize the number of times a vertex is loaded into the cache. Since every vertex is loaded at least the first time it is accessed, we would ideally like to render all its incident triangles before it is flushed out. For a constant size (say, k) cache, we cannot satisfy this for all the vertices simultaneously [Bogomjakov and Gotsman 2001]. This gives rise to the *Face Reordering Problem* [Hoppe 1999]: Given T, the set of triangles in the input mesh, find a permutation of the triangles $\pi_0(T)$, such that $C(\pi_0(T)) \leq C(\pi(T))$, where $C(T)$ is the

number of cache misses for T, and $\pi(T)$ is any of the $m!$ valid permutation of the m triangles in T. Bar-Yehuda et al. [Bar-Yehuda and Gotsman 1996] present an algorithm that generates $n(1 + \frac{c}{k})$ cache misses, (for a constant c), and runs in $O(\frac{n^2}{k^2})$ time. The complexity categorization of the *Face Reordering Problem* is an open question.

Recall that we aim to find a representation that not only reduces the number of cache misses, but also reduces the size of data sent to the graphics card. It turns out that by solving the former, we get a representation that gives a simple scheme to compress the indices. These indices can be decompressed in the hardware. To enable this, we replace T by T', where $T' = T \bigcup X$, where X is a set of degenerate triangles. A degenerate triangle is formed from two or fewer unique vertices. Further, to avoid any line artifacts, we may form a degenerate triangle by replicating a vertex thrice.

2.1 Reducing Cache Misses

Our algorithm, instead of ordering triangles like many others, directly orders vertices first to optimize cache usage. It then reorders the triangles so that the resulting traversal introduces the vertices into the cache in that given order. Our final representation is similar to generalized triangle strip [Chow 1997]. However, we use the actual cache size, k, to locally optimize the length of the strips to try to retain a vertex in cache until all its incident triangles have been rendered. We will argue later that the knowledge of the actual cache size is not crucial for our approach to work well. The *reaccess independent* [1] nature of FIFO aids in finding such an order. This solution is only locally optimal, but in practice it does approach the $n(1 + \frac{1}{k})$ bound [Bar-Yehuda and Gotsman 1996] for the cache misses.

The following example demonstrates our underlying approach. Consider the regular mesh shown in Figure 1, which is replicated from [Hoppe 1999]. Let us divide it into rows of vertices (numbered $V_1..V_r$) (Figure 2). The triangles are also numbered in a row major fashion, with triangles between rows i and $i+1$ forming row T_i of triangles. A total of $r-1$ rows of triangles are formed.

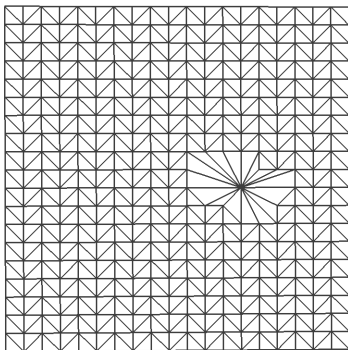

Figure 1: Grid20: a mesh with 391 vertices and 704 triangles.

First, assume the cache is larger than the number of vertices in any two consecutive rows. Now, say, we render the rows of triangles in increasing order. The triangles in a row are issued so that the order in which the vertices enter the cache is similar to the order in which they appear in that row. For example, in row V_1: $V_{1,5}$ is issued before $V_{1,6}$. Note, though, that T_1 uses vertices from both V_1 and V_2, and hence as a boundary condition, we must 'warm-up' the cache with the vertices in row V_1 issued as degenerate triangles. Now when triangles in row T_1 are issued from left to right, vertices in row V_2 will be issued from left to right too. Note that we have not reloaded any vertex so far, and none of the vertices of the first row are flushed out. After the row T_1, the triangles in row T_2 are issued. This brings into the cache, the vertices in V_3 in a similar fashion, while potentially flushing out the vertices of V_1. Hence, we are able to render the whole mesh without reloading any vertex. Of course, we allowed a potentially large cache size. For a fixed cache size k, we adapt our solution as follows. We cut the mesh into several submeshes by subdividing along the width, and perform one pass of the algorithm described above for each cut. If the row-width of each cut is smaller than k -1, we do not reload any vertex for the first pass. On the other hand, the last vertex of each row will be required again for the second cut, and hence potentially result in cache misses. Figure 6 shows these cuts. For this example, we subdivide the mesh into *two* cuts with *eighteen* vertices being reloaded.

Finally, let us generalize to non-regular meshes. For any general mesh, the degree of the vertices may vary substantially. This dictates the way we choose our cuts: we want to minimize the number of edges having endpoints in different cuts, as well as the number of cuts. This general algorithm is phrased in terms of the discussion above:

- Form rows of vertices (*chains*) for the mesh.
- Order the vertices within each chain.
- Form cuts of vertices for each row in accordance with the connectivity information and the cache size k.
- Form a list of triangles (including degenerate triangles) that preserves the vertex order while they are loaded into the cache.

Each step is explained in detail next.

2.1.1 Forming Chains (Rows of Vertices)

We expect the input mesh to be a 2-manifold (with/without boundary). A non-manifold mesh can be broken down into manifolds [Szymczak et al. 2001; Touma and Gotsman 1998].

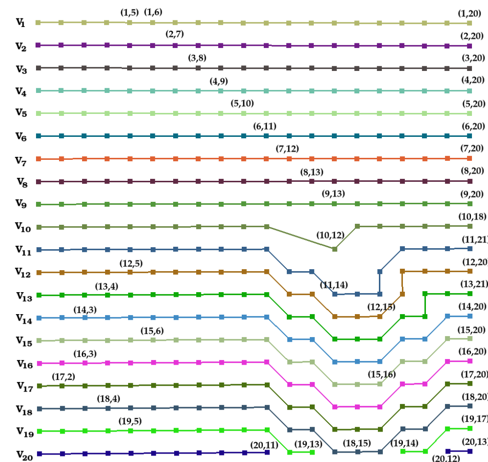

Figure 2: Ordering of the vertices for Grid20. The index of some of the vertices is shown. Vertices with the same color belong to the same chain.

[1] In FIFO, the priority of a vertex is determined by the time of its introduction into the cache. Thus, even if a vertex is accessed repeatedly from a cache, its priority does not change. Hence a vertex remains in the cache for a fixed number of cache miss events. This fact is exploited by our algorithm as we try to load all neighboring vertices of a particular vertex before the subsequent (fixed number of) cache misses occur.

Given the set of all vertices, V, consider a subset V_1, such that vertices in V_1 form a connected path. As an example, say we choose the vertices $V_{1,1}..V_{1,20}$ in Figure 2. Now perform a Breadth First Search (BFS) of the input mesh and find all the vertices in V that are connected to at least one vertex in V_1. Each level of the BFS search would form a chain. Some of these chains may not form a connected path. In this example, we find all the vertices in V that are connected to at least one vertex in V_1. Denote this set as V_2. Continue forming such sets so that each vertex in V_i is connected to at least one vertex in V_{i-1}. Say we form l such sets of vertices, $V_1..V_l$, which represent the l chains in the model. The total running time of this step is $O(m+n)$. The choice of the starting path is arbitrary. For example, V_1 could be assigned the vertex with the minimum or maximum degree. Figure 3 shows the different chains for different starting heuristics. The figure on the top left corner has the top row of vertices as the first chain, while the top right corner has the starting chain as the vertex with the minimum degree. Both these figures depict the Grid20 model. The figures on the bottom of Figure 3 use the vertex with the maximum degree as the initial chain. Note that near the boundaries of meshes, these chains may not form a connected path (V_{20} of Figure 2). Let $s_i = |V_i|$ and $s_{max} = \max_j(s_j)$.

2.1.2 Vertex Ordering within each chain

Having obtained the chains of vertices, we now impose an ordering within each chain. Start with V_1. Since it forms a path, an ordering is already implicit. For each vertex v in V_2, store P_v, the ordered list of IDs of vertices in V_1 that have an edge with v. Now sort (in increasing order) the vertices in V_2, with P_v as the key. In case of ties (i.e. two or more vertices in V_2 having the same set of edge neighbors in V_1), we maintain the order imposed by the edges interconnecting such vertices. This defines the local order of such vertices. We further look at the previous and next set of neighboring vertices (if available) to choose the first (or respectively last) vertex amongst such sets. The rest of the vertices follow the local order just defined. We continue iterating over all rows. Thus, the vertex order in V_i determines the ordering of V_{i+1}. Figure 2 shows the ordering of vertices for our example. The running time for this step is $O(ls_{max}\log(s_{max}))$, which for certain extreme meshes could be $O(n\log(n))$.

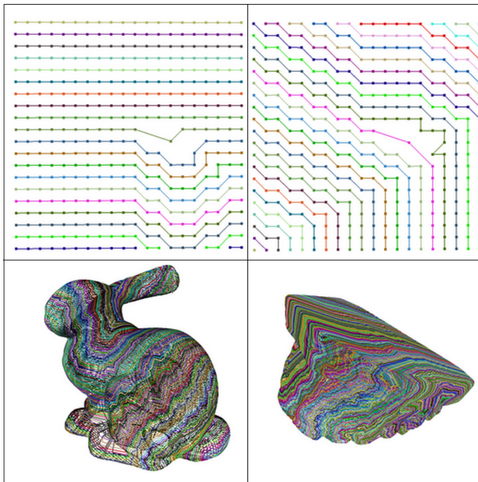

Figure 3: Vertices with the same color belong to one chain. Top: Different rows of vertices using different starting heuristics for Grid20. Bottom Left: Rows of vertices (chains) generated for the Bunny model. Bottom Right: Chains generated for the teeth model.

2.1.3 Forming Cuts in each chain

A *cut* is defined as a subset of vertices within each chain. Recall that each vertex of the mesh can be indexed using a pair of indices (i,j), where i refers to the chain number and j is the index within the chain. For a vertex v , we refer its vertex neighbors (within the cut) belonging to the same chain but having an index less than it as it as its left neighbors. The right neighbors are similarly defined. The top neighbors refer to its vertex neighbors belonging to the previous $((i-1)^{th})$chain. The bottom neighbors belong to the next $((i+1)^{th})$ chain. We use the term row and chain interchangeably.

The subset of triangles having all their end points within these subchains would be reordered in such a way that the subset of vertices (i.e., those included in the cut) in the ith chain are introduced into the cache before the corresponding subset of vertices in the $(i+1)^{th}$ chain. Moreover we need to ensure that none of these vertices are reloaded (while rendering the cut). Hence the following properties need to be satisfied:

1. A vertex v stays in the cache until all its right and bottom neighbors have been loaded in the cache. Hence all its adjacent triangles would be rendered before it is discarded.
2. A vertex v is included in the current cut if all its left neighbors (if it has any) satisfy Property 1.
3. However, for a cache size k, a vertex v is included only if:
   (a) Not more than k vertices are chosen for any chain.
   (b) The number of unique bottom neighbors of vertices before it in the chain is less than k.
4. The sum of the number of unique bottom neighbors of vertices before and including v and the number of right neighbors of v should be the largest possible value less than k.

Property 1 ensures that all incident triangles to a vertex are rendered before it is flushed out. Property 2 aims to include more vertices in the cache provided Property 3 is satisfied. Property 3(a) respects the cache size limitations while Properties 3(b) and 4 aim towards removing a vertex as soon as all its adjacent vertices have been loaded.

We have devised the following 2-pass linear-time algorithm that accomplishes the above by simulating the cache.

1. We process vertices in their assigned order. Say the current vertex is v. We consider all neighbors $w_1,..,w_{d_v}$ of v. w_i needs to be fetched into cache if any of the two neighboring triangles to the edge $v\bar{w}_i$ have not been rendered so far. Otherwise v does not require w_i to be loaded.
2. After simulating Step 1 for k vertices for a row, we need to find the actual *cut* for that row. We keep a *counter* of the number of vertices that can be loaded for a particular row, initialized to (k-1). When we select the first available vertex, we decrement the *counter* by the number of unique bottom neighbors that this vertex would introduce to the cache. We keep adding vertices to the current cut while the *counter* is $\geq$ 0. We then advance to the next row.

Figure 4 shows a step-by-step movement of adding vertices for a particular row to a cut. Say we have a cache of size 8. When we select the first vertex of row i (Fig. 4(a)), the counter value reduces to 3 (since we need to accommodate this vertex and its 4 bottom neighbors). Now consider the second vertex in row i (Fig. 4(b)). Including this vertex into the cut would introduce just *one* new bottom neighbor, and the counter value would become two. Hence we include this vertex in the current cut, and further advance our cut. All the vertices included in the cut are colored red. The third vertex

(Fig. 4(c))introduces *two* new bottom neighbors, which reduces the counter value to 0, thereby defining the final set of vertices from row i (three in total) for the cut.

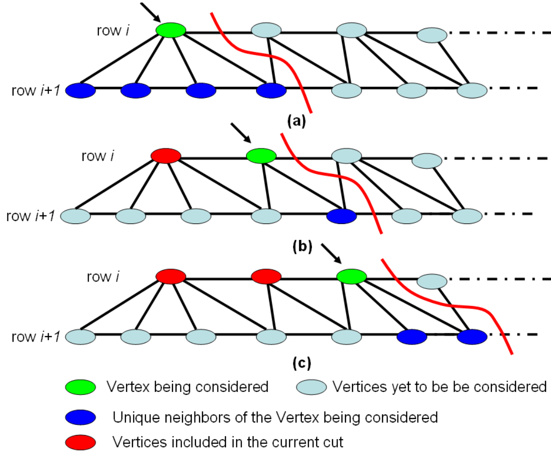

Figure 4: Choosing vertices on a row to locally maximize the cache utilization.

Our algorithm makes progress by choosing vertices from each row so that at least one triangle is formed between consecutive rows. This is ensured even in the presence of vertices with large degrees (by replicating such handful of vertices and sharing the connectivity amongst them). In Figure 5, we show the cut for two consecutive rows for different cache sizes. Our algorithm gives the maximum number of vertices that can be loaded into a cache so as to satisfy the four properties listed above. We mark all the vertices for which all adjacent unmarked vertices have been considered in this cut. For every row, the starting vertex for each cut is chosen as the *first* unmarked vertex in the *previous cut* for that row. We also mark all the edges whose all incident triangles have been considered. We form these cuts until all the edges have been marked.

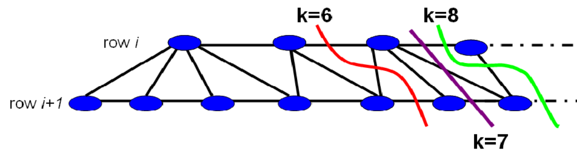

Figure 5: Different cuts for varying cache sizes, satisfying the mentioned properties.

Definition: A vertex is said to lie on the boundary of a *cut* if not all its neighboring vertices lie inside the same cut. A vertex is said to lie *strictly inside* a cut if all its neighboring vertices lie inside the same cut.

It is easy to show that none of the vertices *strictly inside* a cut are reloaded. Further, say, a vertex v lies in c_v cuts (typically c_v is 1). The number of cache misses for a mesh is bounded by $\Sigma_v c_v$.

The running time of this step is $O(n+m)$. In Figure 6, we show the cuts obtained for different models. On the top-left of Figure 6, we show the triangle mesh, while we show the smooth shaded triangles for the other cases.

2.1.4 Triangle Index Generation

The cuts of vertices dictate the order in which the vertices need to be added to the cache. The final step derives an order of triangle indices that respects the vertex order.

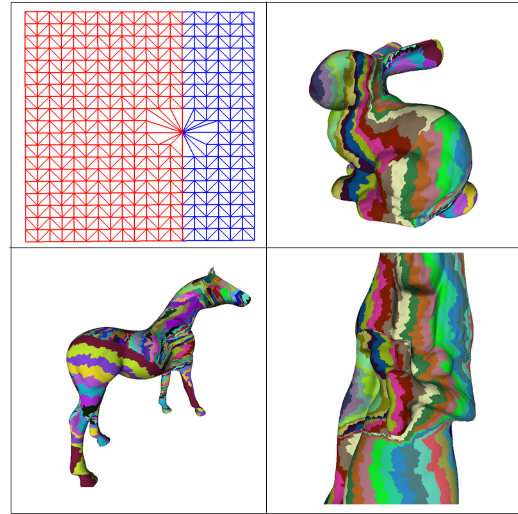

Figure 6: Cuts formed for a cache size of 16 on different models. All triangles belonging to a cut are drawn with the same color.

We use a simulated cache, $\mathscr{C}$, to aid this step. Let $V_1..V_r$ be the sequence of unloaded vertices for a particular row. Let V_{head} be the first unconsidered vertex, and V_{head+j} the jth vertex after it in this sequence. Let $Tri(v_1, v_2, v_3)$ denote an input triangle formed by v_1, v_2 and v_3 (in any order) and let $Out(v_1, v_2, v_3)$ be the triangle indices returned. Let $\mathscr{C}_i$ denote the ith entry in the cache. We perform the following steps:

1. $V_{head} \leftarrow V_1$.
2. If $\exists Tri(V_{head}, \mathscr{C}_x, \mathscr{C}_y)$, $Out(\mathscr{C}_x, \mathscr{C}_y, V_{head})$ and goto Step 4.
3. If $\exists Tri(V_{head}, V_{head+1}, \mathscr{C}_x)$, $Out(\mathscr{C}_x, V_{head}, V_{head+1})$, and goto Step 4. Otherwise goto Step 5.
4. While $\exists Tri(\mathscr{C}_x, \mathscr{C}_y, \mathscr{C}_z)$, $Out(\mathscr{C}_x, \mathscr{C}_y, \mathscr{C}_z)$. Goto Step 6.
5. $Out(V_{head}, V_{head}, V_{head})$. Goto Step 6.
6. Go back to Step 2 with V_{head} reassigned as the first unloaded vertex in that list. If no such vertex exists, exit.

The above generates a list of triangles (with indices). This forms our $\pi_0'(T)$, with a running time of O(n k log k).

We also re-index these indices in the order of their first occurrence in the list of triangles. Hence the first triangle has indices 0, 1 and 2, and we sequentially reassign all the unique vertices. The vertex data is also reordered to reflect the new index positions. This ensures that the vertex attributes associated with these triangles are also in close proximity of each other; thereby reducing the time taken to load them (exploiting the cache of the memory hierarchies) and also aiding in vertex attribute compression using offset based quantization schemes.

The total running time of the complete algorithm, driven by the vertex ordering step, is O(n log n).

3 Topology Coding

Triangle strips are effective topology encoders, but do not necessarily exhibit good cache performance [Diaz-Gutierrez et al. 2006]. They are also difficult to derive from triangle soups. Alternatively, one could subdivide models into small vertex arrays, and hence use short indices. However, smaller the vertex array, larger the vertex replication across them. This increases the size and reduces cache

coherence. We propose a technique to obtain topology encoders even more compact than strips while retaining the close to optimal cache performance described above.

We base our representation on vertex arrays, reducing the size of each index to approximately $\log(k)$ bits. We also present a conceptual architecture to decompress it in hardware.

3.1 Compression Scheme

Let r represent the cache miss ratio. Assume n $\sim \frac{m}{2}$. While rendering the m triangles, $3m$ indices are issued. $\frac{1}{2}m$ vertices of the input model each incur a cache miss the first time they are issued. Another $(r-\frac{1}{2})m$ vertices also incur a cache miss and need to be reloaded into the cache. The remaining $(3-r)m$ indices can be retrieved from the cache without incurring a cache miss. Typically r $\sim$ 0.6, (Table 1), hence a large fraction of indices can be retrieved from the cache itself. Let us represent each of these $k+2$ scenarios for the next loaded vertex with different symbols, namely $(\tilde{1},\tilde{2},..,\tilde{k},\tilde{F},\tilde{R})$, where the first k symbols represent the position in the cache, $\tilde{F}$ represents the first instance of that vertex, while $\tilde{R}$ represents the case where the vertex is reloaded after an earlier flush. We also have an associated probability with each symbol. Hence, we can use Huffman coding, which gives the minimum storage size (within a bit of the optimal value [Cover and Joy 1991]). Let the corresponding codes be $(H_{\tilde{1}},H_{\tilde{2}},..,H_{\tilde{k}},H_{\tilde{F}},H_{\tilde{R}})$. Let $(p_{\tilde{1}},p_{\tilde{2}},..,p_{\tilde{k}},p_{\tilde{F}},p_{\tilde{R}})$ be the corresponding probabilities of occurrence of the symbols.

The list of $3m$ indices are compressed as follows. Here f refers to a mapping function that maps a triangle index to the time-stamp of its first access. In other words, it reorders the vertices in the order they will be first issued. f is filled up during the execution of this algorithm. Also, we maintain a global counter g (initialized to 0). For each index I, one of the following three cases occur:

1. If I is encountered for the first time, then $f(I) = g$. Increment g. Output $H_{\tilde{F}}$.
2. Otherwise, if I is in cache (at position j), then output $H_{\tilde{j}}$.
3. Otherwise, output $H_{\tilde{R}}$ followed by $f(I)$.

The total length of the compressed stream equals
$3m$ $(\sum_{i=1}^{k} p_{\tilde{i}}|H_{\tilde{i}}|+p_{\tilde{F}}|H_{\tilde{F}}|+p_{\tilde{R}}(|H_{\tilde{R}}|+\log(n)))$, where $|H_{\tilde{j}}|$ is the length of the jth codeword.

Since for Case 3 we store a unique global ID for the reloaded vertex, it takes $\log(n)$ bits. Even if the probability of this case is low, dependence on n may be undesirable. An alternative is to exploit coherence and encode only the index offset from the most recently reloaded vertex. Recall that vertices lying along the boundary of the cut need to be generally reloaded. When a new cut is processed, the offset from the top of the previous cut may be large. In order to bound this offset, we need to bound the number of rows (*chains*) that are traversed in a cut before the next cut is processed. This can be achieved by subdividing the rows of the mesh into patches.

3.2 Decompression Scheme

Let h_{max} = $\max(|H_{\tilde{1}}|,|H_{\tilde{2}}|,..,|H_{\tilde{k}}|,|H_{\tilde{F}}|,|H_{\tilde{R}}|)$. For our scenario, typically $h_{max} = \log(k)+1$. We exploit the suffix free property of the Huffman codes to construct a lookup table consisting of $2^{h_{max}}$ entries. With each entry, we store the actual code, and also its length. We keep a global counter g (as during the compression stage, initialized to 0). We also maintain an internal cache (simply an array, with k elements). This need not be the actual vertex cache. (In principle, it need not even be of the same size as the vertex cache - it just needs to match the size used at the compression time. Of course, since our coherence enhancing scheme is based on the real vertex cache size, the compression and cache utilization are maximized if the two cache sizes are the same.) The decompression cache simply stores the index of the vertex at each entry.

Let us assume that we assign IDs in order: 0 .. k -1 for the first k symbols, $(\tilde{1},\tilde{2},..,\tilde{k})$, k to the symbol $\tilde{F}$ and $(k+1)$ to $\tilde{R}$. The decompression algorithm has the following steps.

1. Read the next h_{max} bits from the compressed stream.
2. Compute the relevant symbol ID, S, from the Huffman table, and push the extra bits back to the input stream.
3. If S $< k$, it corresponds to a cache location, output the index Cache[S].
4. If S $= k$, output g. Increment g.
5. If S $= k$+1, extract $\log(g)$ bits from the stream, and output the resulting number.

3.3 Extensions

In order to improve triangle assembly speed, we propose another scheme where a list of triangles is specified using an array of numbers - each number fully encoding a triangle. Each triangle's case now is a triplet of symbols: $\{L_1,L_2,L_3\}$. These symbols may now be Huffman encoded. On the other hand, variable bit operations may be expensive to implement in the hardware. So, we finally propose an algorithm that uses a fixed length for each triangle.

In the following description, instead of using each cache-location as a symbol, we create a new symbol $\tilde{C}$ to mean that a vertex is "in cache". If we consider the three indices of a triangle together, we can eliminate a few cases. In particular, we never allow $\{\tilde{R},\tilde{R},\tilde{R}\}$. If none of the vertices are in the current cut, the triangle should have been drawn in a previous cut. Further, when symbols with multiple cache entries, (e.g. $\{\tilde{C},\tilde{C},\tilde{C}\}$ occur), we have less entropy. We know that three entries must be different. This symbol has $\binom{k}{3}$ possible cache locations instead of k^3, which was inherent in the independent index scheme.

In order to adapt our scheme to a fixed length coding per triangle, we first eliminate larger codes. Since $\tilde{R}$ symbols require the most bits for the specification of the vertex address, we never allow a triangle with two $\tilde{R}$ symbols. Since the fraction of $\tilde{R}$ is small, the probability of $\{\tilde{R},\tilde{R},\tilde{?}\}$ is even smaller, usually less than 0.1% (see Table 3). We eliminate these cases by introducing a degenerate triangle with a vertex corresponding to the first $\tilde{R}$ symbol (thus turning it into a $\tilde{C}$). We label this case with $\{D,D,\tilde{R}\}$.

We classify the remaining cases into the following types: $\{\tilde{F},\tilde{F},\tilde{F}\}$, $\{\tilde{F},\tilde{F},\tilde{C}\}$, $\{\tilde{F},\tilde{F},\tilde{R}\}$, $\{\tilde{F},\tilde{C},\tilde{C}\}$, $\{\tilde{F},\tilde{C},\tilde{R}\}$, $\{\tilde{C},\tilde{C},\tilde{R}\}$, $\{\tilde{C},\tilde{C},\tilde{C}\}$, The first *three* bits of triangle code encode these labels. The rest of the bits hold the address of the $\tilde{C}$ or $\tilde{R}$ vertices. Although, it is possible to encode many triangles in less than 8 bits, allowing up to 16 bits per triangle works for all cases and cache sizes up to 64. Note that due to our increased coherence, the most frequent symbol is $\{\tilde{C},\tilde{C},\tilde{C}\}$. By postponing a triangle with this symbol until one or more of its vertices need to be flushed, we force one of its addresses to be among the front three of the FIFO queue, requiring only 2 bits to address. We can reduce this further by realizing that $\{\tilde{F},\tilde{F},\tilde{F}\}$ is a rare label and the only one requiring a flush of three vertices from the cache. We eliminate it by converting it into a degenerate $\{\tilde{F},\tilde{C},\tilde{C}\}$ followed by a $\{\tilde{F},\tilde{F},\tilde{C}\}$. Now, all $\{\tilde{C},\tilde{C},\tilde{C}\}$ labels point to at least one of the first two cache entries. We split these two cases into labels $\{\tilde{C},\tilde{C},\tilde{C}0\}$ and $\{\tilde{C},\tilde{C},\tilde{C}1\}$. Only the addresses of two cache locations need to be stored now and we are back to eight labels.

By our cluster splitting we ensure that most of our $\tilde{R}$ offsets are small. The very few that do not fit in 13 bits, we turn into $\tilde{F}$ cases by replicating all vertex attributes. This allows us to keep the decompression simple. The largest remaining code is $\{\tilde{C}, \tilde{C}, \tilde{R}\}$, which uses 3 bits for the case mask and $2\log(k) - 1$ bits for the two cache addresses. For our goal of 16 bits per triangle, this still leaves insufficient space for some $\tilde{R}$ indices (for any reasonable k). When 16 bits are inadequate, we introduce a degenerate $\{D, D, \tilde{R}\}$ triangle as described above. The original $\{\tilde{C}, \tilde{C}, \tilde{R}\}$ triangle now becomes $\{\tilde{C}, \tilde{C}, \tilde{C}0\}$ or $\{\tilde{C}, \tilde{C}, \tilde{C}1\}$. This scheme ensures that the most significant *three* bits act as a multiplexer into case decoders. The rest of the vertex address computation logic is straightforward as well.

4 Results and Analysis

We tested our algorithm on a variety of models, ranging from a few thousand triangles to a million triangles. The details of the models are given in Table 1.

Model	*Vertices (n)*	*Triangles (m)*	*Cache Miss (r)*	*Lin et al. (r)*	*Degen. Tris (d)*
Grid20	391	704	0.580	0.605	0.029
Fandisk	6,475	12,946	0.588	0.595	0.024
Bunny	35,947	71,884	0.608	0.597	0.036
Horse	48,485	96,966	0.589	0.599	0.026
Teeth	116,604	233,204	0.590	0.604	0.029
Igea	134,556	269,108	0.573	0.601	0.023
Isis	187,644	375,284	0.580	0.603	0.034
Hand	327,323	654,666	0.612	0.606	0.047
Tablet	539,446	1,078,890	0.567	0.580	0.023

Table 1: Details of the models used for experimentation. The values of r and d are for a FIFO cache of size 16.

In column 4, we report our cache miss ratio r for a cache size of 16 (the typical size of the vertex cache in modern graphics cards [Nvidia 2001]). In column 5, we report the corresponding numbers using the best known results [Lin and Yu 2006]. As evident from these numbers, our algorithm outperforms the best known results for almost all the models (except bunny and hand), even though our focus is on a compression friendly topology representation. Our r values are also *better* than any other reported values in the literature [Hoppe 1999; Nvidia 2004]. Our algorithm reduced the cache miss rate by more than 25% over the widely used NvTriStrip library [Nvidia 2004]. This increased the rendering rates by around 10% - 15% as compared to their representation on several nVidia chips. The number of degenerate triangles (column 6) generated reduces with the increase in cache size, and is *never* more than 5% of the original triangles for all cache sizes (≥ 16). Our preprocessing times vary from a few seconds (for smaller models) to a maximum of few minutes on models consisting of over million triangles. Note that this reordering is needed just once in the lifetime of a static model.

Recently [Yoon et al. 2005] described an algorithm for cache oblivious layouts of meshes. It is worth pointing out that although we optimize our layout for a given cache size, it adapts well to other cache sizes also. It is so because of the inherent property of our algorithm that it attempts to maximize the locality of vertex reference. In Figure 7, we compare the ratio of cache misses for our layout (optimized for a target cache size of 16) against the cache oblivious layout. For cache sizes less than 16, the cache misses never exceed by more than 9%. Since our layout is optimized for a cache size of 16 (in this figure), it performs the best at that size. It outperforms the oblivious layout for a cache sizes up to 64, after which both taper off to the amortized value of 0.5. The lesson is that for smaller (and practical) cache sizes, it pays to use an estimate of

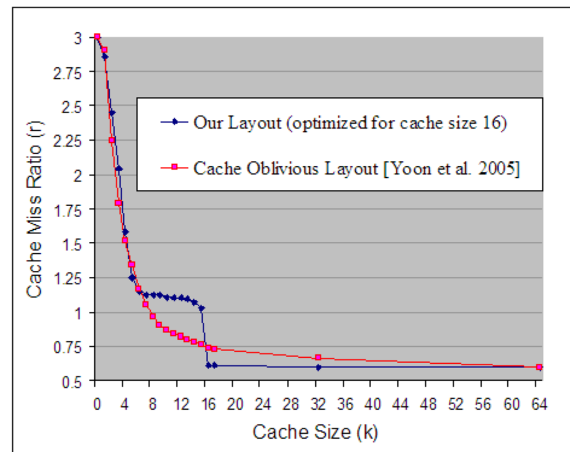

Figure 7: Comparison of the cache misses of our algorithm with the cache oblivious layout for varying cache sizes.

the size. Truly size-oblivious algorithms tend to perform relatively poorly for these practical sizes.

In Figure 8, we plot the average cache miss ratio for various models with varying FIFO cache sizes. For an infinitely large regular triangulation, the number of cache misses has a lower bound of $\mathrm{n}(1+\frac{1}{k-1})$. We get $r = \mathrm{n}(1+\frac{c}{k-1})$, where $2 \leq c \leq 4$ for almost all the models we ran our algorithm on. Hence our approximation algorithm seems to approach the theoretical bounds (*within small constant factors*) for all cache sizes. This makes it applicable for a wide variety of models and graphics cards. In particular, for the tablet model, c is close to 2. This is attributed to the regular structure of the mesh as compared to the bunny model, whose irregular mesh structure results in c being closer to the upper bound of 4.

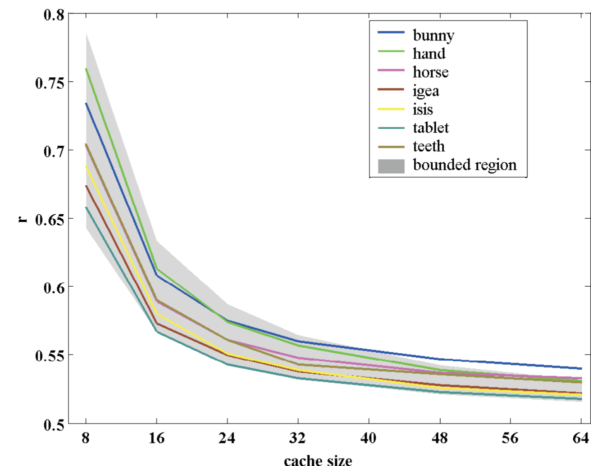

Figure 8: A graph showing the average cache misses incurred by different models with varying cache sizes.

In Table 2, we show the results of our compression algorithm on different models with varying cache sizes. Here, each index is compressed separately. Most of the models have similar compression results, which is a function of the cache size and the ratio of cache misses.

In Table 3, we show the details of per-triangle encoding. Column 2

Model	*8*	*16*	*32*	*64*
Fandisk	4.078	4.491	5.161	5.941
Bunny	4.512	4.681	5.235	5.943
Horse	4.372	4.580	5.172	5.909
Teeth	4.438	4.615	5.186	5.898
Igea	4.347	4.547	5.140	5.864
Isis	4.414	4.587	5.148	5.858
Hand	4.857	4.809	5.278	5.929
Tablet	4.355	4.551	5.130	5.846

Table 2: Average number of bits per index using Huffman codes for each index for varying cache sizes.

shows the percentage of $\{\tilde{C}, \tilde{C}, \tilde{R}\}$ cases in various models. It can be seen that around 5% of the triangles have this label. We also report the percentage of $\{\tilde{C}, \tilde{R}, \tilde{R}\}$ cases, which are indeed low for all the models. In column 4, we give the average number of bits necessary per triangle, which is usually around 8.

Model	$\tilde{C}\tilde{C}\tilde{R}$%	$\tilde{C}\tilde{R}\tilde{R}$%	*bits per triangle*
Horse	6.2	0.10	8.276
Teeth	6.2	0.12	8.292
Igea	5.1	0.02	8.454
Isis	5.6	0.01	8.186
Hand	7.6	0.09	8.534
Tablet	4.5	0.02	8.032

Table 3: Per-Triangle Encoding Results.

Each consecutive pair of triangles also exhibit a high degree of coherence. Around 70% of triangles have at least one index common with its previous neighbor, while around 40% have two of their indices common. Incorporating these into our compression scheme further reduces the compressed representation to around 7 bits per triangle. The hardware friendly mesh representation proposed by [Deering 1995] uses around 40 bits per triangle, while the corresponding numbers reported by [Chow 1997] are around 20 bits per triangle. Thus our representation obtains a 2-4X reduction in the bandwidth over the best known results.

Compared to triangle strip formation algorithms, our algorithm not only results in fewer cache misses, but also produces better compression. The naive approach of subdividing an input mesh into small patches and forming strips in each is not competitive. Although that approach can have similarly low storage requirement for vertex indices, the duplication of vertices across patches is substantial (around 35% to 40% to achieve near 8 bits of storage per triangle), thus blowing up the vertex attribute data itself. In contrast, our algorithm does not increase the original vertex count and still provides a cache friendly compressed representation of the input geometry.

5 Conclusion

Our algorithm produces a compression of mesh topology such that the resulting data can be efficiently decompressed on hardware. Furthermore, the sequence of triangles in the compressed stream is cache-coherent, so that the results of vertex shading may be re-used. We have demonstrated a simple scheme that compresses models to 8-9 bits per triangle on average using variable length encoding. In addition, 16 bits per triangle are sufficient if fixed length codes are desired. Our algorithm to increase cache coherence obtains cache hit ratios quite close to the theoretically maximum possible for constant sized caches. Experiments show that the cache reuse of our representation compares favorably to that of algorithms optimized only for cache coherence. At the same time the compression provided by our algorithm compares favorably with those of other hardware friendly topology compressors.

References

ALLIEZ, P., AND DESBRUN, M. 2001. Progressive compression for lossless transmission of triangle meshes. In *Proc. ACM SIGGRAPH*, ACM, 195–202.

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

BOGOMJAKOV, A., AND GOTSMAN, C. 2001. Universal rendering sequences for transparent vertex caching of progressive meshes. In *Proc. Graphics Interface*, ACM, 81–90.

CHOW, M. M. 1997. Optimized geometry compression for real-time rendering. In *Proc. IEEE Visualization*, IEEE, 347–354.

COVER, T. M., AND JOY, A. T. 1991. *Elements of Information*. John Wiley and Sons.

DEERING, M. F. 1995. Geometry compression. In *Proc. ACM SIGGRAPH*, ACM, 13–20.

DIAZ-GUTIERREZ, P., BHUSHAN, A., GOPI, M., AND PAJAROLA, R. 2006. Singlestrips for fast interactive rendering. *Visual Computer 22*, 6 (June), 372–386.

ESTKOWSKI, R., MITCHELL, J. S. B., AND XIANG, X. 2002. Optimal decomposition of polygonal models into triangle strips. In *Proc. Symposium on Computational Geometry*, ACM, 254–263.

EVANS, F., SKIENA, S. S., AND VARSHNEY, A. 1996. Optimizing triangle strips for fast rendering. In *Proc. IEEE Visualization*, IEEE, 319–326.

GANDOIN, P. M., AND DEVILLERS, O. 2002. Progressive lossless compression of arbitrary simplicial complexes. In *Proc. ACM SIGGRAPH*, ACM, 372–379.

GUEZIEC, A. P., BOSSEN, F., TAUBIN, G., AND SILVA, C. 1999. Efficient compression of non-manifold meshes. In *Proc. IEEE Visualization*, IEEE, 73–80.

GUMHOLD, S., AND STRASSER, W. 1998. Real time compression of triangle meshes. In *Computer Graphics*, IEEE, 133–140.

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

IDSOFTWARE, 2005. http://www.idsoftware.com.

ISENBURG, M. 2001. Triangle strip compression. *Computer Graphics Forum 20*, 2, 91–101.

KRONROD, B., AND GOTSMAN, C. 2002. Optimized compression of triangle mesh geometry using prediction trees. In *Proceedings of 1st International Symposium on 3D Data Processing Visualization and Transmission (3DPVT-02)*, IEEE, 602–608.

LIN, G., AND YU, T. P. Y. 2006. An improved vertex caching scheme for 3d mesh rendering. *IEEE Transactions on Visualization and Computer Graphics 12*, 4, 640–648.

NVIDIA, 2001. D3d optimization: http://developer.nvidia.com/attach/6523.

NVIDIA, 2004. Triangle strip library:developer.nvidia.com/object/nvtristrip_library.html.

PURNOMO, B., BILODEAU, J., COHEN, J. D., AND KUMAR, S. 2005. Hardware-compatible vertex compression using quantization and simplification. *ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware 2005*.

ROSSIGNAC, J. 1999. Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 5*, 1, 47–61.

SZYMCZAK, A., KING, D., AND ROSSIGNAC, J. 2001. An edgebreaker-based efficient compression scheme for regular meshes. *Comput. Geom. Theory Appl. 20*, 1-2, 53–68.

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

TOUMA, C., AND GOTSMAN, C. 1998. Triangle mesh compression. In *Proc. Graphics Interface*, ACM, 26–34.

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*, ACM Press, ACM, 71–78.

YOON, S.-E., AND LINDSTORM, P. 2006. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics 12*, 5, 47–61.

YOON, S.-E., LINDSTROM, P., PASCUCCI, V., AND MANOCHA, D. 2005. Cache-oblivious mesh layouts. *ACM Trans. Graph. 24*, 3, 886–893.