Random-Accessible Compressed Triangle Meshes

Sung-Eui Yoon, *Member, IEEE*, and Peter Lindstrom, *Member, IEEE*

Abstract—With the exponential growth in size of geometric data, it is becoming increasingly important to make effective use of multilevel caches, limited disk storage, and bandwidth. As a result, recent work in the visualization community has focused either on designing sequential access compression schemes or on producing cache-coherent layouts of (uncompressed) meshes for random access. Unfortunately combining these two strategies is challenging as they fundamentally assume conflicting modes of data access. In this paper, we propose a novel order-preserving compression method that supports transparent random access to compressed triangle meshes. Our decompression method selectively fetches from disk, decodes, and caches in memory requested parts of a mesh. We also provide a general mesh access API for seamless mesh traversal and incidence queries. While the method imposes no particular mesh layout, it is especially suitable for cache-oblivious layouts, which minimize the number of decompression I/O requests and provide high cache utilization during access to decompressed, in-memory portions of the mesh. Moreover, the transparency of our scheme enables improved performance without the need for application code changes. We achieve compression rates on the order of 20:1 and significantly improved I/O performance due to reduced data transfer. To demonstrate the benefits of our method, we implement two common applications as benchmarks. By using cache-oblivious layouts for the input models, we observe 2–6 times overall speedup compared to using uncompressed meshes.

Index Terms—Mesh compression, random access, cache-coherent layouts, mesh data structures, external memory algorithms.

1 Introduction

Among the key challenges in visualization is how to effectively manage, process, and display large geometric data sets from scientific simulation, computer-aided design, and remote sensing. Today's unstructured meshes measure hundreds of millions of elements and require gigabytes of storage, often greatly exceeding available memory and rendering resources. A compounding factor to this problem is the increasing mismatch between processing performance and the rate at which data can be fed to the CPU and GPU, which is limited by latency and bandwidth [40]. As a result, multilevel caching schemes are commonly employed, with successively smaller but faster caches that provide reduced latency. Such caching schemes are effective as long as there is a reasonably close match between data organization and access patterns. Since access patterns on meshes are usually localized, recent work in the visualization community has focused on coherent organization of meshes and other data [4, 5, 7, 11, 16, 27, 34, 41, 42].

Data compression is a complementary approach to reducing bandwidth requirements. Whereas mesh compression has traditionally been used to reduce on-disk storage or transmission time over slow networks, recent work has explored the possibility of trading underutilized computing power for higher effective disk bandwidth through on-line compression [6, 29].

For large data sets, the access pattern of the application also significantly influences its performance. The concept of *windowed stream processing* was recently proposed for I/O-efficient access to large compressed meshes [19]. However this approach requires restructuring the computation to match the data layout, which is not always possible or even desirable, e.g. when direct access to small subsets of the data is needed. The main competing approach is to keep a raw on-disk mesh data structure that supports random access [9, 10, 18, 28, 36, 42]. However such data structures usually require a significant amount of disk space: up to 40 times more space than compressed meshes [19]. The added bandwidth requirements of such verbose representations often negate the benefits of organizing the meshes to support random access.

Unfortunately, combining compression with coherent data layout is nontrivial. Conventional mesh compression schemes [1, 31, 38] maximize compression by reordering the data as a canonical permutation, which destroys any layout designed for cache coherence. Streaming mesh compression [20] avoids such reordering by also encoding the layout, but restricts decompression to sequential access: to access an element late in the stream, the entire stream up to that element must be decoded, which can be prohibitive for large files. To address this, new schemes have emerged that support selective access to small patches of the compressed mesh [8, 23]. However, these methods are mainly designed for rendering applications; they do not preserve the mesh layout nor support seamless mesh access across patch boundaries.

Main results: In this paper we present a new order-preserving triangle mesh compression algorithm that supports random access to the underlying compressed mesh. Our method selectively fetches, decompresses, and caches the requested parts of the mesh in main memory, possibly after paging out data not recently accessed. The compressor preserves the original, possibly cache-coherent triangle layout, and hence allows optimizing the layout for different modes of access—even for sequential streaming computations. Our decompressor provides direct access to individual vertices and triangles via their global indices in their respective layouts, and exposes to the visualization application a conventional mesh data structure API for transparent access to mesh elements and their neighborhoods. Although we do not maintain the entire mesh and full neighboring information in main memory, we ensure that correct connectivity is constructed for all mesh elements requested by the application. Using layouts with good locality, we achieve compression ratios around 20:1 and speedups as high as 6:1 in out-of-core visualization applications, compared to accessing the same uncompressed external memory data structure. In particular, we find that cache-oblivious mesh layouts [41, 42] result in good compression, high disk cache utilization and thus a small memory footprint, high decompression throughput, and good locality for lower-level in-memory caching compared with other tested layouts.

• *Sung-Eui Yoon is with the Korea Advanced Institute of Science and Technology (KAIST), E-mail: sungeui@cs.kaist.ac.kr.*
• *Peter Lindstrom is with the Lawrence Livermore National Laboratory, E-mail: pl@llnl.gov.*

Manuscript received 31 March 2007; accepted 1 August 2007; posted online 2 November 2007.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

2 Related Work

In this section we review published work related to mesh compression and compression methods that support random access.

2.1 Mesh Compression

Mesh compression has been well researched over the last decade and excellent surveys are available [2, 13]. At a high level, connectivity compressors may be classified as vertex-based (or valence-based) [1, 38], edge-based [21, 26], or face-based [14, 31], depending on the mesh element type that drives the compression. Our order-preserving method belongs to the class of face-based compressors.

Most previous mesh compression schemes were designed to achieve maximum compression as they were targeted for archival use or transmission. They achieved this goal by encoding vertices, edges, and faces in a particular order agreed upon by encoder and decoder such that the mesh layout itself would not have to be transmitted. Because many applications are not affected by the ordering of mesh elements, such reordering is often acceptable.

Recently, Isenburg et al. [20] introduced a streaming compression scheme for triangle meshes built on top of their streaming mesh representation [19]. This compression method efficiently handles massive models by directly encoding mesh elements in the order in which they arrive, which obviates having to first create a complete uncompressed mesh data structure to support traversal of the mesh in the designated order. Our work is built on top of this streaming compression method in order to both preserve the input order of triangles and to achieve relatively high compression and decompression throughput.

2.2 Compression and Random Access

Most prior approaches to mesh compression do not directly provide random access to the compressed data. To access a particular element, the mesh must first be sequentially decompressed to an uncompressed format (e.g. an indexed mesh format like PLY, or a mesh data structure such as half-edge) that supports random access.

Multimedia and regular grids: Random access is one of the key components of the MPEG video compression format that allows users to browse video in a non-sequential fashion [25]. This is achieved by periodically inserting “intra pictures” as access points in the compressed stream, which allows bootstrapping the decompressor. Intra pictures are compressed in isolation from other frames, and subsequent frames are compressed by predicting the motion in between these intra pictures. For regular volumetric grids, wavelet-based compression methods [17,30] that support random access have been proposed.

Mesh and multi-resolution compression: Ho et al. [15] describe an out-of-core technique that partitions large meshes into triangle clusters small enough to fit in main memory, which are compressed independently. Cluster boundaries are given special treatment to ensure that decompressed clusters can be “stitched” together. Choe et al. [8] proposed a random-accessible mesh compression technique primarily targeted for selective rendering. As in [15], random access to the compressed mesh is achieved by independently decompressing a requested cluster, i.e. without having to decompress the whole mesh. There have been a few multi-resolution compression methods that support random access. Gobbetti et al. [12] proposed a compressed adaptive mesh representation of regular grids for terrain rendering. They decompose the regular grid into a set of chunks and apply wavelet-based lossy compression to each chunk. Kim et al. [23] introduced a random access compression technique for general multi-resolution triangle meshes based on their earlier multi-resolution data structure [24].

Although these techniques provide coarse-grained random access to compressed meshes, they are mainly targeted for selective access in rendering applications, and do not provide a general mesh traversal mechanism. On the other hand, our method transparently supports random access to individual mesh elements and provides the connectivity information needed by many mesh processing applications.

2.3 Cache Coherence

Cache-oblivious layouts of polygonal meshes and bounding volume hierarchies have recently been introduced [41–43]. Contrary to cache-aware layouts, e.g. [7,16,27,34], a cache-oblivious layout is not optimized for a particular size cache, but exhibits good average-case performance across multiple cache levels and access patterns. By maintaining cache-oblivious layouts of triangles and vertices in our compressed meshes, we achieve high cache utilization both to compressed data fetched from disk and to uncompressed in-memory data.

3 Overview

In this section we briefly discuss some of the challenges of dealing with massive models and present an overview of our approach.

3.1 Dealing with Massive Models

Applications such as iso-contouring and geodesic distance computation require random access to mesh geometry and connectivity. Such applications access vertices and triangles in an order that generally differs from the order in which the mesh is stored. For large meshes, the amount of information accessed may approach giga- or even terabytes of data. As a consequence, large meshes are usually stored on disk, or may even be fetched over a network. Since accessing remote data in an arbitrary order can be very expensive, data access time often becomes the major performance bottleneck in geometric applications.

Out-of-core data structures and algorithms: There have been extensive research efforts to design out-of-core data structures and algorithms to handle models larger than main memory [37]. These techniques aim at loading only the data necessary to perform local computations, and at minimizing the number of I/O accesses. However, as the gap between processing time and data access time increases, the time spent loading even the necessary data on demand becomes expensive.

Cache coherence: Since cache misses in the various memory levels (e.g., L1/L2 and main memory) are quite expensive compared to the computational processing time, research has focused on reorganizing the data access pattern of applications (e.g. [3,19,39]) and on computing data layouts (e.g. [41–43]) to minimize the number of cache misses for coherent but unspecified access patterns. In particular, data layout optimization can result in high cache utilization without having to modify the algorithm or access pattern of the target application, whereas computational reordering usually requires complete algorithm and data structure re-design.

3.2 Our Approach

We propose a novel compression and decompression method that supports transparent and random access to compressed meshes suitable for many geometric applications. Whereas we support truly “random” access to any element of the mesh, we exploit the fact that most geometric applications access the mesh in a spatially coherent manner, e.g., by walking the mesh from an element to one of its neighbors. However, we neither assume nor impose any particular access pattern. At a high level, our method has two major components: (1) cluster-based order-preserving mesh compression, and (2) a runtime decompression framework that transparently supports random access.

Cluster-based order-preserving compression: We compress a mesh by sequentially accessing and grouping triangles in the order they appear in the input mesh. Although our method does not require a specific layout of a mesh, we propose to use cache-oblivious layouts since they have exhibited superior cache utilization in a number of applications [41]. We also find that these cache-oblivious layouts result in the best compression and runtime performance.

In order to provide random access to the compressed mesh, we group vertices and triangles into a set of *clusters*. Each cluster consists of a fixed number of consecutive triangles (e.g. a few thousand) as well as the vertices first referenced by those triangles. The beginnings of the clusters serve as access points in our method, and each atomic I/O request operates at cluster granularity.

Runtime decompression framework: Our decompression framework provides efficient, but transparent and random access to applications through a general mesh access API. Therefore, applications can access the entire mesh as if it were memory resident. Moreover, applications benefit directly from the improved I/O performance provided by our decompressor without having to make any application code changes since we provide a complete system for loading, decompressing, caching, and paging-out of data. We assume that applications access mesh vertices and triangles via a global index through our mesh access API. When an element is requested, we efficiently locate the cluster containing it, decompress the cluster, and store the uncompressed data in main memory. As the compressed stream does not explicitly encode full connectivity information, we dynamically derive such data on the fly during decompression and link together adjacent in-memory elements in the mesh.

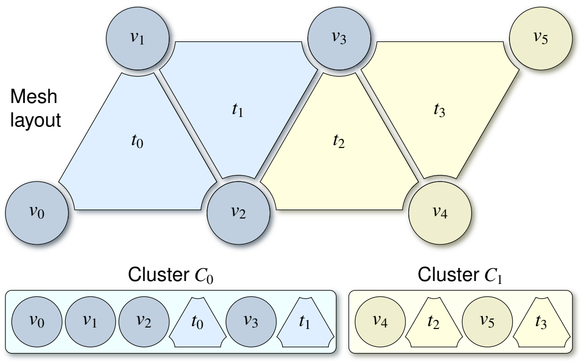


Fig. 1. **Clustering of Vertices and Triangles:** The interleaved sequence of vertices and triangles is *pre-order vertex-compact* [19]. v_i denotes the geometry of the i^{th} vertex; t_i denotes the three vertex indices of the i^{th} triangle. The boxes indicate the decomposition of the mesh into clusters, here consisting of two consecutive triangles.

4 Triangle Mesh Compression

In this section we describe our cluster-based order-preserving compression method. We first review the streaming, sequential access compression scheme that our order-preserving method is built upon. Then, we describe the extensions necessary to support random access.

4.1 Streaming Mesh Compression

Isenburg et al. [20] proposed a streaming compression method for triangle meshes represented in a streaming format [19]. This method sequentially compresses a mesh of nearly arbitrary size in an I/O efficient manner. There are two major components of this method that our compressor also utilizes.

Input data format: The streaming mesh compressor operates on *vertex-compact* streams in *pre-order* format (Fig. 1). In a pre-order format, each vertex appears in the stream before all triangles that reference it. If the stream is also vertex-compact, the first triangle that follows a vertex is guaranteed to reference that vertex. Vertex-compactness ensures that vertex and triangle layouts are interleaved, and that vertices are not buffered earlier than necessary. The requirement that the input be vertex-compact and pre-order is not particularly restrictive as all triangle layouts have a "compatible" (though not unique) vertex-compact pre-order vertex layout. Note that all face-based compressors naturally produce vertex-compact pre-order output.

Finalization: The other key feature of the streaming mesh compressor is the use of *finalization* information. Finalization of a vertex indicates that it will not be referenced by any subsequent triangles in the stream. Hence, during compression we may safely limit references to the set of *active* vertices that have been introduced but not yet finalized. Typically the active vertices are only a small fraction of all vertices, which aids in efficient coding of vertex references. Finalization is usually known to mesh writing applications, and can easily be incorporated with most mesh formats. See [19] for how to compute such information in case it is not readily available.

Our random access compression method uses the pre-order format as input and preserves the order during compression. It also exploits and encodes finalization. Because of this, our compressed meshes can be used also in streaming computations with little overhead, although our method is mainly designed for random (non-sequential) access.

4.2 Cluster-Based Order-Preserving Compression

Our compression method reads intermixed sequences of vertices and triangles in a pre-order streaming file format with finalization information. During compression, we implicitly decompose vertices and triangles into a set of *clusters* (Fig. 1). A cluster C consists of a fixed number of consecutive triangles (e.g., 4K triangles) and those vertices introduced (first referenced) by the triangles in C. As a consequence, although each cluster has a fixed number of triangles, the number of vertices per cluster may not be the same (though the variation in vertex count is usually low).

We compress the triangles and vertices assigned to a cluster only based on information collected from the cluster. Therefore, at runtime each cluster may be independently decompressed, which accommodates random access to the mesh at the granularity of clusters. Note that not all vertices referenced by the triangles in a cluster have their geometry encoded in the same cluster, nor are all triangles incident on a vertex stored in the same cluster. We will resolve this "stitching" problem in our runtime decompression framework.

Terminology: For a mesh element e such as a vertex or a triangle, we represent its index as $Idx(e)$. Let $C(e)$ denote the cluster containing e, and let C_i indicate the cluster whose index is i, with C_0 being the first cluster. If a triangle of C_j references a vertex stored in C_i, we say that C_j references C_i. Let $R_{\prec}(C)$ denote the set of clusters referenced by cluster C and $R_{\succ}(C)$ denote the set of clusters that reference C. Whenever C_j references C_i, we have $C_i \in R_{\prec}(C_j)$ and $C_j \in R_{\succ}(C_i)$.

Cluster properties: Given vertex-compact pre-order input, the following lemmas are easily derived from our definition of clusters. The lemmas will be used later to show the correctness of our method.
Lemma 1 (Triangle existence): For a vertex v introduced in a cluster $C(v)$, at least one triangle in $C(v)$ references v.
Lemma 2 (Triangle containment): The triangles incident on a vertex v are either in the same cluster, $C(v)$, or in $R_{\succ}(C(v))$.

4.3 Encoding Compression Operators

For each cluster, our compressor sequentially reads vertices and triangles from the streaming mesh. For each triangle t, we first determine the compression operator associated with t, which tells how t is connected to the set of already compressed triangles within the cluster.

As in [20], we use five different compression operators (or configurations): Start, Add, Join, Fill, and End. Start indicates that t shares no edge with the already compressed triangles in the cluster; in Add and Join there is one shared edge; in Fill and End there are two and three shared edges, respectively. For example, the sequence of compression operators for the mesh in Fig. 1 is "Start, Add, Add, Add." For Start, Add, and Join cases, we also determine how many new vertices the triangle introduces and encode their geometry. The number of introduced vertices ranges from 0 to 3 for Start, equals 1 for Add, and is 0 for Join. These compression operators can be easily determined by maintaining a *half-edge* data structure. For each coded triangle, three half-edges are created. Once a vertex is finalized, the half-edges incident on the vertex may be deallocated.

Compression side: We compute these sequences of compression operators based on the half-edge data maintained from the first triangle of the input to the current triangle being compressed. This means that we do not deallocate the existing half-edge data when we transit from one cluster to the next during compression. The main reason for this is to avoid any duplicate storing of vertex geometry in the compressed mesh, since otherwise we would not know whether a vertex first referenced in a cluster was introduced here or by a triangle in some earlier encoded cluster. We therefore encode the compression operator in the context of global information of all the encoded triangles and vertices.

Decompression side: In contrast to the compressor, the decompressor does not need to maintain global information for all encoded triangles and vertices. Given a decoded compression operator, we can deduce the vertex indices associated with the triangle. Some of these indices may refer to vertices stored in another cluster. If an application requests geometry (as opposed to only connectivity) information for such vertices, our runtime framework determines which cluster has that information, decodes it, and returns it to the application.

4.4 Encoding Mesh Elements

Once the compression operator for a triangle is encoded, we encode the vertices referenced by the triangle. For each such vertex v, there are two cases: v is referenced for the first time, or v has already been introduced. When v is introduced by a triangle, we encode its geometry and attributes (e.g., color). Note that in this case we do not need to encode v's index since it must equal the current global vertex count. This vertex count is made available to the decompressor by

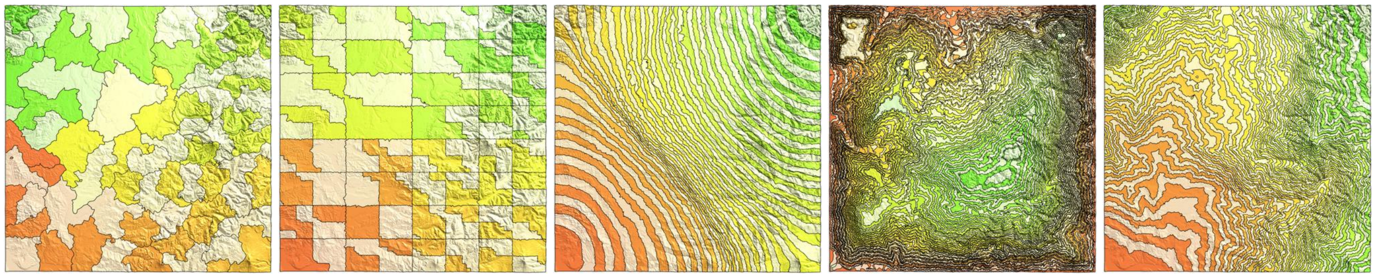

(a) Cache-oblivious layout (b) Z-order space-filling layout (c) Spectral layout (d) Depth-first layout (e) Breadth-first layout

Fig. 2. **Clusters for Different Layouts:** This figure highlights clusters of 8K consecutive triangles for different layouts of the Puget Sound terrain simplified to 512K triangles. The cluster colors smoothly vary with the sequential layout from red to yellow to green, and the brightness alternates between each consecutive pair of clusters. The cache-oblivious mesh layout has high spatial coherence, leading to well-shaped clusters with short boundaries and few inter-cluster references. As a result, it yields the highest compression ratio and best runtime performance on our benchmarks.

storing with each cluster, C, the global index of C's first vertex, which is maintained incrementally during compression.

In the other case, when vertex v was introduced earlier, we are interested only in encoding its index $Idx(v)$ since v's geometry has already been encoded. In order to effectively encode this index, we make use of three layers of compression contexts: (1) a cache holding the three vertices of the previous triangle in the current cluster, $C(v)$, (2) the active (unfinalized) vertices of $C(v)$, and (3) the vertices among the clusters $R{\prec}(C(v))$ referenced by $C(v)$. The first and second layers are for *in-cluster* vertices stored in the current cluster, and the third layer is for *out-of-cluster* vertices stored in other clusters.

If vertex v was also referenced by the previous triangle we encode which of the three vertices in the cache v corresponds to. Otherwise, we check whether v is an active vertex of the current cluster, and if so encode its position in a dynamic vector (containing active vertices) maintained by both encoder and decoder. Note that the set of active in-cluster vertices is usually much larger than the three vertices stored in the cache, but is also much smaller than the entire set of vertices stored in the current cluster. Finally, if v is not among the active vertices, we conclude that it is in another cluster C_i among the set $R{\prec}(C(v))$. In this case, we decompose its index into a pair index (i,k) where i is the global cluster index for C_i and k is an offset within C_i. Instead of directly encoding the pair (i,k), we map the global index i to a local index j within the set $R{\prec}(C(v))$. This is beneficial as the number of clusters in $R{\prec}(C(v))$ is usually much smaller, e.g. 4 on average for cache-oblivious layouts, than the total number of clusters.

After finishing compressing the mesh, we have accumulated information specific to each cluster C, such as C's position in the compressed file, its first vertex index, and the variable-size sets $R{\prec}(C)$ and $R{\succ}(C)$ that must be written as header information. We store this header information as a separate, uncompressed file. In our benchmarks, the header files are roughly 2 MB, or about 1% of the total compressed file size. The decompressor is initialized with this header information to allow any cluster to be decompressed at runtime.

Memory usage and time complexity: The data structures needed to perform the operations described above are small because the data is limited to the set of active vertices and the elements of the current cluster. The time complexity of encoding and decoding a triangle is constant. This is made possible by using a hash table of active vertices to map a global index $Idx(v)$ to the cluster-offset pair (i,k).

Half-edge based coding: We further improve the compression ratio by encoding some of the vertex indices based on existing half-edge information around vertices, as proposed in [20]. We encode the index of an in-cluster vertex by specifying which of the set of half-edges it is associated with. For example, when the compression operator is ADD, we may encode two active vertices by the single half-edge that joins them, and to which the triangle being encoded is attached. Recall that we deallocate all the existing half-edge information when we encounter a new cluster in the decompressor. Therefore, the compressor, too, needs to consider only those half-edges created in the current cluster. This half-edge based coding requires a small amount of computational overhead, such as traversing half-edges for the vertices of a triangle. However, we can achieve a higher compression ratio since the number of half-edges around a vertex is typically small (e.g., 6).

Geometry prediction: We use the parallelogram rule [38] to predict the position of a vertex v_3 introduced by a triangle t. To perform the parallelogram prediction in an ADD operation, we require geometry information for the three vertices of an adjacent, already compressed triangle that shares v_1 and v_2 with t. However, these vertices may be stored in other clusters and, thus, their geometry information may not be available when we compress or decompress the current cluster. In this case, we simply use v_1 or v_2 as prediction, if available, or otherwise the third "opposite" vertex. When no nearby vertex is available, we use the last encoded vertex as prediction.

Arithmetic coding: We use context-based arithmetic coding for compression operators, vertex indices, and geometry. In particular, we use the previous compression operator as context for the next one. We also re-initialize all probability tables with a uniform distribution when compressing a new cluster in order to allow independent decoding of clusters. To minimize the impact of this periodic initialization on compression, we employ a fast adaptive probability modeler [35].

5 Runtime Mesh Access Framework

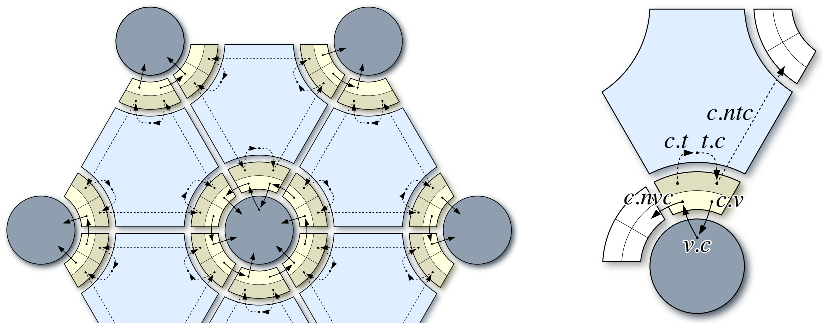
In this section we present our runtime decompression and caching method that provides transparent random access.

5.1 In-Core Mesh Representation

When the application requests geometry or connectivity for a mesh element, our underlying decompression framework fetches and decompresses the cluster containing the element into an in-core partial mesh representation. To support a general mesh access mechanism, we represent our decompressed in-core mesh in a corner data structure similar to the ones proposed by Rossignac [32] and Joy et al. [22]. Conceptually, this data structure consists of two contiguous global arrays of vertices and triangles large enough to hold the entire mesh.

A *corner* associates a triangle t with one of its vertices v (see Fig. 3). For each vertex v we store its coordinates and an index $v.c$ to one of its incident corners. A triangle is represented as three corners that each store an index $c.v$ to the corresponding vertex v and an index $c.nvc$ within a circular linked list to the next corner incident to v. Similarly, pointers $t.c$ and $c.t$ between corners and triangles and pointers $c.ntc$ within triangles allow instant navigation between adjacent elements. By traversing the $c.nvc$ pointers around v, we can find all the triangles incident to v (whether v is manifold or not). As in [32], the corners of triangle i have consecutive indices $3i$, $3i+1$, and $3i+2$. Hence $t.c$, $c.t$, and $c.ntc$ can be efficiently computed and need not be stored.

The corner table can be incrementally constructed via constant-time insertions. As we sequentially decompress the global vertex indices of each triangle, we compute corresponding corner indices from the triangle index. We then insert each corner into its vertex's circular corner list. Because each vertex is introduced by a triangle via a compression operator, at least one incident corner (triangle) is always available.



```
struct Vertex {
  struct Coords {
    float x, y, z;   // geometry
  };
  Index c;           // incident corner
};

struct Triangle {
  struct VertexIndices {
    Index v[3];      // vertices
  };
  struct CornerIndices {
    Index nvc[3];    // next vertex corners
  };
};
```

Fig. 3. **Corner Representation:** A corner c (yellow) associates a triangle t (light blue) with one of its vertices v (dark gray). For each corner c, we provide access to the next corner $c.ntc$ in t, as well as the next corner (in no particular order) $c.nvc$ around v. These corner pointers form circular linked lists around vertices and triangles. We also store a pointer $c.v$ to v from each incident corner c, and a pointer $v.c$ from v to one of its corners. Similar pointers between corners and triangles, as well as $c.ntc$, can be derived on the fly, and need not be stored (shaded/dotted). The actual vertex and triangle data structures are shown on the right.

5.2 Mesh Access API

We provide the following atomic API to support random access to the compressed mesh based on the data structures in Fig. 3:

Coords **GetVertex**(Index v_{Idx}): Return the coordinates of vertex v.

VertexIndices **GetTriangle**(Index t_{Idx}): Return the three vertex indices of triangle t.

Index **GetCorner**(Index v_{Idx}, Index t_{Idx}): Return the corner joining vertex v with triangle t.

Index **GetVertexCorner**(Index v_{Idx}): Return one of the corners, $v.c$, incident to v.

Index **GetTriangleCorner**(Index t_{Idx}): Return one of the corners, $t.c$, of triangle t.

Index **GetCornerVertex**(Index c_{Idx}): Return the vertex, $c.v$, associated with corner c.

Index **GetCornerTriangle**(Index c_{Idx}): Return the triangle, $c.t$, associated with the corner c.

Index **GetNextVertexCorner**(Index c_{Idx}): Return the next corner, $c.nvc$, incident on the vertex associated with corner c.

Index **GetNextTriangleCorner**(Index c_{Idx}): Return the next corner, $c.ntc$, within the triangle associated with corner c.

Based on this low-level API, it is possible to implement higher-level functionality. For example, to compute all the triangles incident to a vertex, we make a call to GetVertexCorner followed by a sequence of interleaved GetCornerTriangle and GetNextVertexCorner calls. We implement our benchmark applications, discussed later, using our API.

5.3 Page-Based Data Access

Whenever a request to access a mesh element is made, we have to first identify the cluster containing it. Though clusters have a fixed number of triangles (and thus corners), their vertex counts generally vary. Therefore mapping vertices to clusters is not straightforward, and techniques like binary search can be slow for large meshes. Since every vertex access requires a cluster lookup, e.g. to determine whether the cluster is cached, it is important that this lookup be done efficiently.

To provide a fast mechanism for mapping vertex indices to clusters, we decompose the global vertex array into fixed-size contiguous *pages*, each of which holds a power-of-two (e.g., 1K) vertices (Fig. 4). With each page we store the memory address of the corresponding vertex sub-array, the indices of the clusters that overlap the page, as well as a state variable that indicates whether the page is cached and, if so, the level of connectivity information available: "none," "partial," or "full" connectivity (to be explained in Sec. 5.4). We keep similar cluster-specific state with the in-core cluster meta data, and each page's state indicates the least common information available for its clusters. The page table is initialized by reading the header file containing cluster file offsets and dependencies $R{\prec}$ and $R{\succ}$ and by marking all pages as "not loaded." This initialization task takes only tens of milliseconds in our benchmarks.

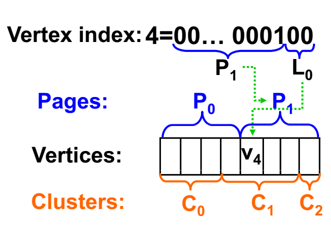


Fig. 4. **Page-Based Data Access:** The figure illustrates our data access mechanism for a page size of four vertices. Given a vertex index, in this case 4, page and local indices are computed via simple bit operations. When page P_1 is requested, we load the overlapping clusters C_1 and C_2 and return the requested vertex v_4 to the application.

The main benefit of this page structure is that it allows accessing the required data using few operations. For example, when GetVertex is called, we first compute the page index corresponding to the vertex using only a bit shift. If the page is not completely loaded, we load all uncached clusters associated with the page, decompress them into our in-core mesh data structure, and set the page's state to "loaded with no connectivity." If the page is loaded the next time it is accessed, we directly return the requested data from our in-core mesh representation, which allows constant-time access to uncompressed data.

One downside of this approach is that more than one cluster may have to be loaded when accessing a single element. On the other hand, if data and access locality are high, it is likely that adjacent clusters will be needed for subsequent vertex accesses. Nevertheless, to reduce the average number of clusters per page to close to one, we make the page size smaller than the cluster size.

5.4 On-Demand Connectivity Construction

To reduce the number of I/O requests, we dynamically construct only the connectivity needed to correctly execute our API calls. As explained earlier, pages and clusters can have "no," "partial," or "full" connectivity, corresponding to the amount of information needed by the calls GetVertex, GetVertexCorner, and GetNextVertexCorner, respectively. These states are described below.

No connectivity: If the page $P(v)$ containing v is loaded as a result of a GetVertex(v) call, no effort is made to compute connectivity information for the page since only v's geometry is needed. In this case, we initialize $v.c$ for all vertices in $P(v)$ to null.

Partial connectivity: To process a GetVertexCorner(v) call, we gather sufficient connectivity information for the page $P(v)$ containing v. We first determine the clusters $C(P(v))$ that overlap $P(v)$ and then visit each triangle t contained in these clusters. For each vertex $u \in t$, we determine whether u is currently cached, and if so connect t to u via a $c.nvc$ corner pointer. Since some vertices referenced by t may reside in other clusters not yet cached, we put any corners corresponding to such uncached vertices on a *SkippedCorners* list stored with t's cluster. This list holds corners not yet connected to their vertices and adjacent corners, and will be consulted later when such uncached vertices are loaded. After we process all the clusters of the page, we set their states and the page's state to "partially connected." At this point, we are guaranteed by Lemma 1 (see Sec. 4.2) to have at least one cached corner for vertex v, which can then be returned by GetVertexCorner.

Full connectivity: When GetNextVertexCorner(c) is called to access the next corner around the corresponding vertex v, *all* the corners incident to v have to be loaded into v's circular corner list in order to guarantee correctness. We achieve this by performing the following steps. We first load the cluster that contains c and identify the vertex v. Then, for each cluster C that overlaps $P(v)$, we load and build partial connectivity for C and the clusters $R_{\succ}(C)$ that reference C. For each cluster $D \in R_{\succ}(C)$, we extract from D's *SkippedCorners* list each corner that corresponds to a vertex $u \in C$ and connect it to u. By Lemma 2 (see Sec. 4.2), after this step all corners around v (and all other vertices in $P(v)$) have been connected, and as a final step we set the state of each $C \in P(v)$ and $P(v)$ itself to "fully connected." The next time GetNextVertexCorner is called, we simply return the corner from our in-core mesh representation if $P(v)$ is "fully connected."

The main difference between the states "partially" and "fully" connected is whether we have to load the clusters $R_{\succ}(C)$ that reference the cluster C containing the requested vertex. By maintaining three separate states, we ensure that the correct results are returned with each API call while maintaining a minimal set of loaded clusters.

5.5 Memory Management

The page table also serves as a memory management mechanism for massive models whose uncompressed data cannot fit in main memory. For this purpose, we also maintain a page table for triangles/corners, with a one-to-one mapping between triangle pages and clusters. Applications may specify a maximum allowance on memory use, which limits the number of pages cached. When a new page is needed and the page table is full, we have to unload a page P and each overlapping cluster C to make room for the new page. Note that vertices in $R_{\prec}(C)$ may have incident corners in C, and thus their full connectivity depend on C being present. To ensure that future connectivity queries to vertices in $R_{\prec}(C)$ are correctly answered, we mark the clusters $R_{\prec}(C)$ as "partially" connected when C is evicted from memory.

We use a FIFO page replacement policy modified as follows for the special case of GetNextVertexCorner calls around vertex v. Before making space in the vertex page table for the uncached cluster $C(v)$ and page $P(v)$, we move all cached pages that overlap $R_{\succ}(C(v))$ to the back of the FIFO, as the triangles in $R_{\succ}(C(v))$ are needed to complete the call. This ensures that the necessary clusters $R_{\succ}(C(v))$ are not evicted as a result of loading $P(v)$ into a full page table.

5.6 Coherent Mesh Layout

Both cluster decompression requests and in-core cache misses can be drastically reduced by organizing mesh vertices and triangles in a coherent order. To achieve this goal, we use *cache-oblivious* techniques [41, 42] to order the triangles of the mesh using the OpenCCL library [44]. We then produce a pre-order vertex-compact layout by reordering the vertices to be "compatible" with the triangles [19], i.e. the vertices are sorted on the order in which they are first referenced by a triangle. While such "induced" vertex layouts are not necessarily optimal, our compressor requires them, and we have empirically observed that they also exhibit good locality [41]. An example of triangle clusters derived from a cache-oblivious layout is shown in Fig. 2.

6 Results

To demonstrate the benefits of our method, we have implemented two applications using our compressed mesh API: iso-contour extraction and mesh reordering. We chose these two applications since they both traverse the mesh in an order that is reasonably coherent though different from the original layout. Moreover, iso-contour extraction typically accesses only a small subset of the mesh, whereas reordering requires traversing the entire mesh.

We have implemented our compressor, decompressor, and applications on an Intel Pentium 4 mobile laptop running Windows with a 2.1 GHz CPU, 2 GB of main memory, and a 15 MB/s IDE disk drive. We limit our applications to use no more than 1.5 GB of main memory to cache uncompressed data. Our compression method requires as input a streaming mesh format, which is straightforward to write or to generate from non-streaming formats [19].

Model	Elements		Raw	Compressed					Speedup	
	#T (M)	#V (M)	Size (MB)	Size (MB)	Header (MB)	Geom. (bpv)	Conn. (bpv)	Ratio	Contour	Reorder
Puget Sound	134	67	3,712	173	2.2	13.5	7.9	21.4	2.5	6.7
RMI isosurface	102	51	2,842	171	1.7	19.4	8.2	16.6	2.3	4.3
St. Matthew	128	64	3,543	178	2.1	14.8	8.1	19.9	2.4	3.4

Table 1. **Compression and Speedup:** Triangle and vertex counts, file size of uncompressed and compressed meshes, compression ratio, and iso-contouring and mesh reordering speedups are listed. The coding cost is separated into geometry, connectivity, and cluster header data. The uncompressed (raw) meshes are stored on disk in our corner table representation (Fig. 3), modified to use 16-bit quantized integer coordinates to match the precision used by the compressor. The meshes were compressed using 4K triangle clusters and a cache-oblivious layout.

Model	Triangles	Vertices	TG [38]	CKLLS [8]	ILS [20]	Ours	Raw
Dino	28,096	14,050	19.8	22.8	25.2	31.8	452.0
Igea	134,342	67,173	17.2	17.7	22.3	25.0	452.0

Table 2. **Compression Comparison:** Mesh size in bits per vertex is reported for four different methods, including ours. For a fair comparison with [8], we use 12-bit quantization and 50 clusters. TG and CKLLS results are excerpted from [8]. Since our method preserves the layout and supports random access, its compression ratio is lower compared to the other techniques.

6.1 Compression Results

We evaluate our compression method on several benchmark models, including a large, simplified terrain model of the Puget Sound area (Fig. 5), the RMI iso-surface model from LLNL, and Stanford's St. Matthew model, each totaling over 100 million triangles (see Table 1). We uniformly quantize each vertex coordinate to 16 bits, which is more than enough precision to faithfully represent these meshes. Our compressor encodes these meshes at an average rate of 380K triangles per second on our laptop. For example, it takes around 6 minutes to compress the Puget Sound model.

We compare the file sizes of our compressed meshes with those of the original uncompressed meshes stored on disk in the corner table representation shown in Fig. 3, modified to use 16-bit integer rather than floating-point coordinates. Though a conventional indexed mesh representation requires less space, it does not support the same functionality required by our API and by our benchmark applications. Also, while constructing a full corner table from an indexed mesh can be done in linear time, performing this task at startup incurs unacceptable overhead and wastes disk space. Compared to the uncompressed corner table representation stored in cache-oblivious order, our compressor reduces the three benchmark models by factors 17–21, and by 9–12 compared to indexed meshes. This results in 21.4, 28.0, and 23.3 bits per vertex (bpv) for the Puget Sound, RMI isosurface, and St. Matthew model, respectively.

Comparison with other methods: Compared with Isenburg and Gumhold's out-of-core compressor [18], which neither preserves the layout nor supports random access, our compressed representation of the St. Matthew model is 50% larger than theirs (15.3 vs. 22.9 bpv). We compare our method against the order-preserving compressor of Isenburg et al. [20]. The overhead in storage incurred by our method relative to theirs is on average a modest 16% for our large models. This overhead is mainly due to additional information (e.g., cross-cluster vertex references) needed to support random access.

We also compare the compression ratio of our method to those of Touma and Gotsman (TG) [38] and Choe et al. (CKLLS) [8]. The overhead of our method is about 40% and 50% over CKLLS and TG (see Table 2). Like [18], the TG method does not support random access. Although the CKLLS method does, it does not accommodate seamless mesh traversals or order preservation for transparent mesh access and higher cache utilization.

6.2 Iso-contouring

The problem of extracting an iso-contour from a scalar function defined on an unstructured mesh frequently arises in geographic information systems and scientific visualization. Many efficient iso-contour extraction methods employ seed sets to grow a contour by traversing

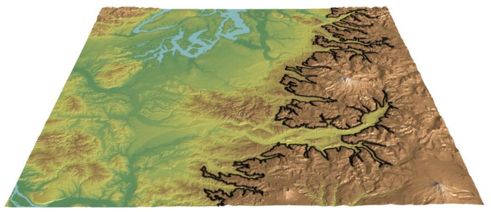


Fig. 5. **Puget Sound Iso-contour:** The contour line (in black) at 720 m elevation was extracted from an unstructured terrain model consisting of 134M triangles. The contour passes through 286K triangles.

only those mesh elements that intersect the contour. The running time of such an algorithm is typically dominated by the traversal of the intersected mesh elements. We efficiently extract an iso-line from a seed triangle by traversing the contour in a depth-first order, thereby accessing the surrounding mesh in a reasonably coherent manner, but in an order different from the layout of the mesh.

We compare the running time of extracting iso-contours for 20 randomly chosen iso-values on the three benchmark models using (1) our compressed representation with 4K triangle clusters and (2) a fully uncompressed on-disk corner table. Both representations are stored in the same cache-oblivious layout and accessed using the same API. (For our non-terrain surfaces, we use one of the coordinates as function value, which reduces iso-contouring to "slicing" the mesh.) We do not perform explicit memory management of the uncompressed meshes, but rely on the virtual memory management of the operating system (which includes disk block buffering) by memory mapping the uncompressed corner table file.

We achieve on average 2.5 times and as much as 6.4 times speedup extracting one iso-contour from the Puget Sound mesh when using the compressed representation. Similar gains are observed on other models (Table 1). The main reason for this speedup is the drastic reduction in expensive disk reads while traversing and loading uncached portions of the mesh. Though compression reduces disk space and data transfer, it increases memory use. We measure the total working set size, i.e. the amount of data loaded and cached, in our application in multiples of the 4 KB memory page size on our system. With a cluster size of 4K triangles and a cache-oblivious layout, our method uses four times as much memory as when no compression is used. The two main reasons for this are: (1) we cache data at a coarser granularity (128 KB clusters versus 4 KB memory pages), and (2) when a cluster C is accessed, our method often requires loading additional clusters $R_{\succ}(C)$ that reference C in order to ensure correct connectivity.

Below we will further discuss the performance of our method in detail using the Puget Sound model as a test case.

Dependence on cluster size: We measure iso-contouring performance and compressed file size as a function of cluster size using cache-oblivious layouts and a fixed vertex page size that on average equals half the cluster size. In general, compression improves with larger cluster size as a result of fewer out-of-cluster references, which impacts both connectivity and geometry rates. Larger clusters also improve I/O throughput because of the size-independent overhead due to disk latency. On the other hand, very large clusters increase the working set size and reduce the ability to selectively access mesh elements, which negatively impact performance.

These competing factors are illustrated in Fig. 6, which shows that the optimal cluster size in terms of overall performance is 4K triangles. This cluster size also results in good compression compared to using much larger clusters. Assuming a 2:1 ratio between triangles and vertices, a 4K triangle cluster decompresses to 128 KB of in-core storage (i.e. irrespective of the compression rate). The corresponding compressed size on disk is 5.3 KB per cluster for Puget Sound.

Dependence on layout: We compare the performance of iso-contouring using compressed and uncompressed meshes in different layouts, including cache-oblivious (COML) [41, 42], Z-curve [33], depth-first (DFL), breadth-first (BFL), and spectral (SL) [19] layouts of the Puget Sound model (see Table 3). Note that the Z-curve is also a cache-oblivious layout that works particularly well for regular grids.

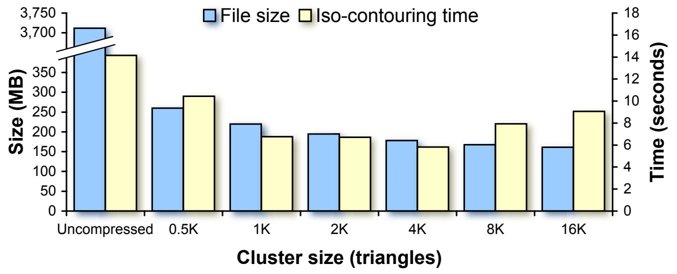


Fig. 6. **Dependence on Cluster Size:** The Puget Sound iso-contouring time and compressed file size depend on the cluster size. The highest performance is attained using 4K triangles per cluster, for which the compression ratio is 21:1 and the speedup is 2.4 relative to using no compression.

Layout	Size (MB)	Contouring time (s)		Speedup
		Raw	Compressed	
Cache-oblivious	178	5.1	2.1	2.4
Z-curve	261	8.2	5.0	1.6
Depth-first	287	8.6	6.0	1.4
Breadth-first	312	62.0	44.9	1.4
Spectral	461	31.0	91.0	0.3

Table 3. **Dependence on Layout:** The table lists the compressed file size and iso-contouring time for various layouts of the Puget Sound model stored in compressed and uncompressed format. The use of compression results in speedups as high as 2.4 for the cache-oblivious layout, which also performs better in relation to all other layouts. Out-of-core access to the spectral layout, whose poor locality leads to low compression and excessive paging, is not accelerated by compression.

Using compression we achieve 2.4, 1.6, 1.4, and 1.4 times iso-contouring speedup for COML, Z-curve, DFL, and BFL, respectively. Although we observe meaningful speedups with depth- and breadth-first layouts, our results show a clear advantage of using cache-oblivious or similar layouts that exhibit spatially coherent clusters, both relative to other layouts (e.g., 4, 16, and 21 times speedup over DFL, BFL, and SL, respectively) and to using no compression. Higher coherence results in a smaller working set size, fewer I/O calls, better in-memory cache utilization, and hence better performance. This ability to optimize and preserve the layout during compression is one of the features that sets our scheme apart from prior methods like [8].

Table 3 shows that compression hurts performance when used with the spectral layout. Though globally coherent, triangles in this layout appear in a nearly random order along the advancing front, which is often wider than the 4K cluster size. This leads to poor compression and locality, and excessive loading of clusters.

The layout of a mesh also significantly affects compression, as is evident from Table 3. Because of its well-shaped clusters, the cache-oblivious layout yields the best compression ratio among our layouts.

Comparison with stream processing: As demonstrated in [19], streaming computations can be very efficient for out-of-core processing of large meshes. We compare our compressed random-accessible meshes with the sequential-access compression scheme of Isenburg et al. [20] by measuring the time to extract an iso-contour from Puget Sound stored in a cache-oblivious layout. Whereas our scheme allows random access to the elements intersected by the contour, the streaming technique supports only sequential access and hence must traverse and decompress (nearly) the entire mesh. As a consequence, we obtain a 45:1 speedup over the streaming scheme on this task.

There are other geometry processing tasks more suitable for streaming access (e.g. smoothing, vertex normal computation) that require only local information around mesh elements, and for which the processing order does not matter (e.g. sequential access is possible). Because our scheme uses streaming (de)compression within each cluster and provides "finalization" information, it also efficiently supports streaming access. Sequential decompression of the entire data set via our API takes only 33% longer than using Isenburg et al.'s scheme. Since we efficiently support both random and sequential access, we believe that our method has a significant advantage over theirs.

6.3 Mesh Reordering

As evidenced here and in [19,37,42], the problem of computing a good layout of a large mesh is itself an important but challenging problem that traditionally is done using external sorts. As another benchmark, we compute a breadth-first triangle layout from a (different) cache-oblivious one, which cannot be done efficiently using external sorts alone. This task differs from iso-contouring in that the entire mesh is traversed, but is similar in that it requires random access and is therefore not easily streamable. Using compression, we achieve 3.4–6.7 times speedup on our benchmark models (Table 1). Moreover, we observe similar speedups using clusters in the range 2K–16K triangles.

7 Conclusion and Future Work

We have proposed a novel out-of-core framework that supports transparent random access to compressed triangle meshes through selective decompression of small clusters of mesh elements. In order to provide a seamless mesh traversal mechanism, our method dynamically constructs the connectivity information necessary for querying incidence and adjacency information through a common mesh access API. One distinguishing feature of our method is that it preserves the ordering of triangles in the mesh, which allows tailoring the data layout to the anticipated access pattern. In conjunction with cache-oblivious layouts, we demonstrate that the reduced I/O bandwidth implied by compression leads to significant improvements in performance without the need for end-application code changes. We show that other layouts also benefit from mesh compression, and that our compressed representation can be used efficiently for sequential stream processing. Source code for our compressor and mesh access API is freely available at http://www.cs.unc.edu/~sungeui/RAC.

We envision many avenues for future work. Foremost, our current scheme is primarily suited for read-only access, and we would like to extend the method to efficiently handle modifications to the mesh, e.g. for geometry processing and interactive editing. The intrinsic partitioning of the mesh into independent clusters suggests the potential for parallel computations. One benefit of our scheme is that it obviates overlapping layers of “ghost” information across clusters. Furthermore, domain decomposition can be efficiently done by assigning clusters to compute nodes. Finally, we plan to investigate extensions of our method to hierarchical data in order to improve the performance of ray tracing and collision detection between massive models.

Acknowledgements

We would like to thank Martin Isenburg for sharing his streaming mesh compression code with us. This work was performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48, and was supported in part by a KAIST seed grant.

References

[1] P. Alliez and M. Desbrun. Valence-driven connectivity encoding for 3D meshes. *Computer Graphics Forum*, 20(3):480–489, 2001.

[2] P. Alliez and C. Gotsman. Recent advances in compression of 3D meshes. *Advances in Multiresolution for Geometric Modelling*, 3–26. 2005.

[3] L. Arge, G. Brodal, and R. Fagerberg. Cache oblivious data structures. *Handbook on Data Structures and Applications*, chapter 34. 2004.

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

[5] A. Bogomjakov and C. Gotsman. Universal rendering sequences for transparent vertex caching of progressive meshes. *Computer Graphics Forum*, 21(2):137–149, 2002.

[6] M. Burtscher and P. Ratanawoabhan. High throughput compression of double-precision floating-point data. *IEEE Data Compression Conference*, 293–302. 2007.

[7] J. Chhugani and S. Kumar. Geometry engine optimization: Cache friendly compressed representation of geometry. *ACM Symposium on Interactive 3D Graphics and Games*, 9–16. 2007.

[8] S. Choe, J. Kim, H. Lee, S. Lee, and H.-P. Seidel. Mesh compression with random accessibility. *Israel-Korea Bi-National Conf.*, 81–86. 2004.

[9] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, 2003.

[10] C. DeCoro and R. Pajarola. XFastMesh: Fast view-dependent meshing from external memory. *IEEE Visualization*, 363–370. 2002.

[11] P. Diaz-Gutierrez, A. Bhushan, M. Gopi, and R. Pajarola. Single-strips for fast interactive rendering. *The Visual Computer*, 22(6):372–386, 2006.

[12] E. Gobbetti, F. Marton, P. Cignoni, M. Di Benedetto, and F. Ganovelli. C-BDAM—Compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum*, 25(3):333–342, 2006.

[13] C. Gotsman, S. Gumhold, and L. Kobbelt. Simplification and compression of 3D meshes. *Tutorials on Multiresolution in Geometric Modelling*, 319–361. Springer, 2002.

[14] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. *ACM SIGGRAPH*, 133–140. 1998.

[15] J. Ho, K. Lee, and D. Kriegman. Compressing large polygonal models. *IEEE Visualization*, 357–362. 2001.

[16] H. Hoppe. Optimization of mesh locality for transparent vertex caching. *ACM SIGGRAPH*, 269–276. 1999.

[17] I. Ihm and S. Park. Wavelet-based 3D compression scheme for interactive visualization of very large volume data. *Computer Graphics Forum*, 18(1):3–15, 1999.

[18] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. *ACM SIGGRAPH*, 935–942. 2003.

[19] M. Isenburg and P. Lindstrom. Streaming meshes. *IEEE Visualization*, 231–238. 2005.

[20] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming compression of triangle meshes. *Symposium on Geometry Processing*, 111–118. 2005.

[21] M. Isenburg and J. Snoeyink. Face Fixer: Compressing polygon meshes with properties. *ACM SIGGRAPH*, 263–270. 2000.

[22] K. I. Joy, J. Legakis, and R. MacCracken. Data structures for multiresolution representation of unstructured meshes. *Hierarchical and Geometrical Methods in Scientific Visualization*, 143–170. Springer, 2003.

[23] J. Kim, S. Choe, and S. Lee. Multiresolution random accessible mesh compression. *Computer Graphics Forum*, 25(3):323–332, 2006.

[24] J. Kim and S. Lee. Truly selective refinement of progressive meshes. *Graphics Interface*, 101–110. 2001.

[25] D. Le Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, 1991.

[26] J. Li and C. C. Kuo. A dual graph approach to 3D triangular mesh compression. *IEEE ICIP*, 891–894. 1998.

[27] G. Lin and T. P.-Y. Yu. An improved vertex caching scheme for 3D mesh rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):640–648, 2006.

[28] P. Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. *ACM Symp. on Interactive 3D Graphics*, 93–102. 2003.

[29] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006.

[30] F. Rodler. Wavelet based 3D compression with fast random access for very large volume data. *Pacific Graphics*, 108–117. 1999.

[31] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, 1999.

[32] J. Rossignac. 3D compression made simple: Edgebreaker with zip & wrap on a corner-table. *Shape Modelling & Applications*, 278–283. 2001.

[33] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[34] P. V. Sander, D. Nehab, and J. Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM SIGGRAPH*, 2007. To appear.

[35] M. Schindler. Range encoder version 1.3, 2000. URL http://www.compressconsult.com/rangecoder/.

[36] E. Shaffer and M. Garland. A multiresolution representation for massive meshes. *IEEE Transaction on Visualization and Computer Graphics*, 11(2):139–148, 2005.

[37] C. Silva, Y.-J. Chiang, W. Correa, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. *IEEE Visualization Course Notes*. 2002.

[38] C. Touma and C. Gotsman. Triangle mesh compression. *Graphics Interface*, 26–34. 1998.

[39] J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[40] W. Wulf and S. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[41] S.-E. Yoon and P. Lindstrom. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1213–1220, 2006.

[42] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. *ACM SIGGRAPH*, 886–893, 2005.

[43] S.-E. Yoon and D. Manocha. Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum*, 25(3):507–516, 2006.

[44] S.-E. Yoon, D. Manocha, P. Lindstrom, and V. Pascucci. OpenCCL, 2005. URL http://gamma.cs.unc.edu/COL/OpenCCL/.