A Survey of Efficient Structures for Digital Geometry Processing

Keenan Crane

April 25, 2006

Contents

1 Introduction

In the past decade massive data sets produced by high-precision scanning and high-fidelity simulation have inspired new data structures and algorithms for efficient digital geometry processing. The main feature of these tools is an increasing trend towards locality, both in the order of processing and in the arrangement of data. The title image shows one of the most common examples of a massive mesh: Michaelangelo's "David," scanned during the Digital Michaelangelo Project [35]. The original dataset consists of approximately two billion triangles, occupying tens of gigabytes of storage space. Even at a reduced resolution of 56 million triangles, this model does not easily fit in the address space of a 32-bit machine. Limited memory resources have motivated the central issue in high performance mesh processing, known as *out-of-core* (OoC) processing.

Traditionally, mesh processing has not focused on data layout. Even today most meshes are stored as an arbitrarily ordered list of triangles and vertices, linked together via indirect references. Using this representation, processing, e.g., the immediate neighbors of a vertex may require access to several different pages in memory or sectors on disk. Not only is this behavior counter-intuitive, it results in unacceptable performance when processing large data sets.

This survey examines the progression of solutions for OoC geometry processing. Although these solutions were developed as a reaction to memory limitations, the same technology applies in the domain of stream processing and parallel processing, as discussed in section 5. Additionally, while most of the discussion focuses on triangle meshes, similar ideas apply to other surface representations, as discussed in section 3.4.

Several other surveys on OoC geometry processing are available, including the course notes from IEEE Visualization 2002 [48]. Surveys related to *in*-core mesh processing are also relevant, such as those by Heckbert & Garland [25], and Garland [18].

2 Background

2.1 Surfaces

Geometry processing tends to focus on two-dimensional *surfaces*, and most algorithms will handle surfaces which are *compact*, *closed*, *connected*, and *orientable* (i.e., anything which has the topology of a sphere or a torus). By "surface" we really mean a *topological surface* with an *embedding* into $\mathbb{R}^3$. A topological surface is a set of points S such that for each point $p \in S$ there exists an open neighborhood containing p and contained by S which is homeomorphic to $\mathbb{R}^2$ (we will call the corresponding homeomorphisms h_p). An embedding of that surface is a homeomorphism $f : S \to \mathbb{R}^3$ which associates a geometry with S. (Note that although the image of f may intersect itself, the underlying space does not.) In essence, we are only interested in the "shell" of an object, rather than what is on the inside. A surface is *smooth* if the composition $h_p^{-1} h_q|_{N(p)\cap N(q)}$ is infinitely differentiable for any two points $p, q \in S$, where $N(x)$ is the neighborhood around the point x for which h_x is defined. A smooth surface can be "bumpy," but it cannot change abruptly (e.g., a golf ball is smooth, but a cube is not).

Many geometry processing algorithms operate on surfaces *with boundary*, meaning that h_p may

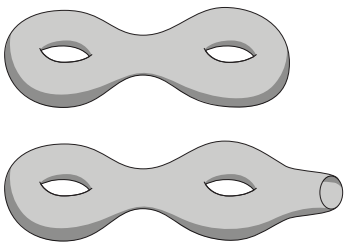

Fig. 1: *Examples of surfaces without (top) and with (bottom) boundary.*

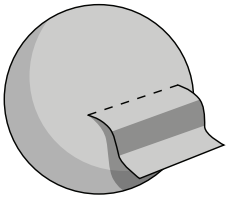

Fig. 2: *Surface is non-manifold along the dashed line.*

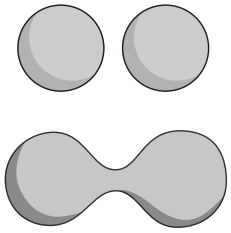

Fig. 3: *A surface with two components (top) and a suface with one component (bottom).*

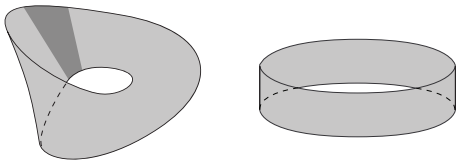

Fig. 4: *A non-orientable Möbius strip (left) and an orientable band (right).*

map to $\mathbb{R}^2$ *or* a closed half-plane. Intuitively, a surface with boundary is a surface with "holes" (figure 1). Some algorithms handle geometry with *non-manifold* regions (figure 2), though many algorithms assume manifold input because the topological structure is simpler. *Connected* surfaces consist of only one *component* or distinct object (figure 3). Processing can usually be done one component at a time, so we focus on connected surfaces. *Orientability* has to do with whether a surface has one or multiple "sides" (figure 4). Non-orientable surfaces are not a major concern in mesh processing: most people could go through their entire life without ever seeing a non-orientable surface, although their existence would be a pitiful one. (Interested readers should consult a text such as Bloch [4] or Hatcher [24].)

2.2 Meshes

2.2.1 Mesh Terminology

To process surfaces on a computer, we must choose a discrete digital representation. *Triangle meshes* are the most common representation in geometry processing, and are very similar to *simplicial* surfaces. A typical example of a mesh is shown in figure 5.

A *k-simplex* is the convex hull of $k+1$ affinely independent points (intuitively, none of the points "line up") embedded in $\mathbb{R}^n$, $n \geq k$. In mesh processing, simplices are more commonly referred to as *vertices* (0-simplices), *edges* (1-simplices), *faces* or *triangles* (2-simplices), and *tetrahedra* (3-simplices). We can parameterize a k-simplex $\sigma = \langle v_0, v_1, \ldots, v_k \rangle$ using *barycentric coordinates*:

$$\sigma = \left\{ p \in \mathbb{R}^n : p = \sum_{i=0}^{k} \alpha_i v_i, \sum_{i=0}^{k} \alpha_i = 1, \forall i \; \alpha_i > 0 \right\}.$$

A *face* of a simplex σ is a simplex formed from a subset of the vertices of σ (e.g., the faces of triangles include vertices, edges, and triangles). A simplicial surface S is a set of simplices such that for every simplex $\sigma \in S$ all faces of σ are in S, every 1-simplex in S is a subset of exactly two 2-simplices in S, and for each 0-simplex $v \in S$ the union of 2-simplices which contain v is homeomorphic to a disk. If we replace the second condition with the requirement that every 0-simplex of S be contained in some 2-simplex of S, then we have a simplicial surface with boundary. The purpose of these definitions is to give the union of simplices in a simplicial surface, called its *carrier* or *underlying space*, the same topological structure as a surface.

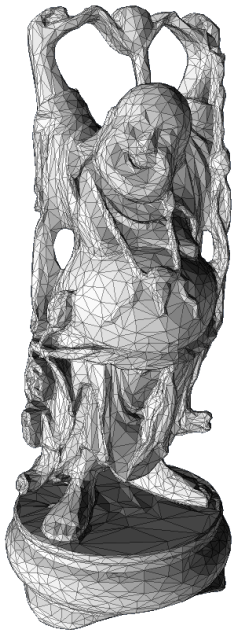

Fig. 5: *"Happy Buddha" Model.*

Whereas simplicial surfaces are collections of 0-, 1-, and 2-simplices, triangle meshes might be collections of 2-simplices only (triangle soup - section 2.2.3), collections of 0- and 2-simplices (indexed meshes - section 2.2.2), or have some more complicated structure (*half-edge* and *quad-edge* meshes - section 2.2.4). The *topology of a mesh* often refers to the connectivity of vertices, faces, etc., which is easily confused with the topology of the corresponding surface (although the two share a number of properties).

When we discretize a smooth surface of high curvature or a surface with many high-frequency features we will have significant *discretization error* because our representation is a piecewise linear reconstruction of the original signal from a sparse set of samples. Discretization error is part of the reason such massive data sets exist: large numbers of small triangles better approximate highly detailed or highly curved surfaces.

Another difficulty is defining attributes of discrete geometry which are good analogues for the corresponding attributes of smooth surfaces. For instance, the actual curvature at any point on a piecewise linear surface is either zero or undefined, but this interpretation is not particularly useful. Instead, we would like a definition of discrete curvature which carries similar semantics and preseves the same invariants as for the smooth case. Precise definition of these relationships is an area of active research - Meyer et al present one framework for differential operators on meshes [40], and Hirani proposes a discrete exterior calculus [26].

2.2.2 Indexed Meshes

```
v -0.5 -0.5 -0.5
v  0.5 -0.5 -0.5
v  0.5 -0.5  0.5
v -0.5 -0.5  0.5
v -0.5  0.5 -0.5
v  0.5  0.5 -0.5
v  0.5  0.5  0.5
v -0.5  0.5  0.5
f 1 2 3 4
f 5 6 2 1
f 6 7 3 2
f 7 8 4 3
f 8 5 1 4
f 5 8 7 6
```

Fig. 6: *Cube represented as a Wavefront OBJ.*

By far the most common mesh representation is an *indexed mesh*, mainly due to the simplicity of writing and parsing such data. This representation is well-encapsulated by the ubiquitous *Wavefront OBJ* format, which stores vertices as ordered triples of coordinates, and faces as ordered tuples of vertex indices. Figure 6 gives an example OBJ file representing a cube. Lines beginning with `v` specify the x, y, and z coordinates of a vertex, and lines beginning with `f` specify the indices of vertices used to construct a face. Vertices' indices are specified implicitly by their order in the file. Although the example shows vertices and faces in contiguous blocks, data may be specified in any order. Note that in this case the faces are rectangles, not simplices.

Unfortunately, index meshes are a particularly treacherous format when processing large meshes: there is no map from the index of a vertex to its location in the file, and the index of a vertex can only be determined by knowing its location in the file. The former makes it impossible for a face to immediately locate one of its vertices, and the latter makes it impossible to search for a vertex intelligently. Consider a file containing n vertices and m faces. If a face needs to locate a vertex with index i, the most efficient scheme will search at least i and at most $m+i$ lines of the file, starting at the beginning. We cannot skip any lines at the beginning of the file, because we need to know how many lines contain vertices in order to track the current index. We cannot even perform, e.g., binary search, because again we have no way to determine the index of the vertex we find, and would not know where in the file to look for subsequent vertices anyway. Other types of queries are equally inefficient. For example, if we want to find all the vertices in the *1-ring* of a specified vertex (i.e., the immediate neighbors of a vertex), we would again have to do a linear search to find all the faces containing the current vertex.

These types of queries would not be a problem if we could load the entire mesh into memory. For example, vertices could be stored in a contiguous array and accessed at random by index in constant time. As mentioned before, however, meshes which are too large to fit in memory are the central problem in high performance mesh processing. Additionally, even meshes which fit in memory will exhibit poor caching behavior if accesses are highly scattered.

2.2.3 Triangle Soup

One way to reduce the indirection used in indexed mesh formats is to store vertices redundantly for each face. For example, a triangle mesh can be stored as a long list of vertices where each triple of consecutive vertices implicitly encodes a triangle, avoiding the expensive search required to form a face. However, finding, e.g., the 1-ring neighbors of a vertex is no easier than before.

Indexed mesh data can be converted into triangle soup by dereferencing or *normalizing* faces' vertex references [8]. The list of triangles in an indexed mesh is sorted three times, once for each vertex. Once sorted, indices can be quickly replaced with actual vertex locations by simultaneously streaming

through the sorted face list and the vertex list (which is sorted by default). For OoC meshes, sorting can be performed with an external sorting algorithm, as discussed in section 2.3.

2.2.4 Half/Quad-Edge Data Structures

Another popular representation for meshes which provides more topological information is the *half-edge* data structure [6]. A simplistic realization of the data structure is illustrated in figure 7: an oriented half-edge he stores a pointer to its originating vertex, $from$, the next half-edge in the face, $next$, and the corresponding half-edge sym in the adjacent face. In this example, faces are stored implicitly and can be found by following the $next$ pointer of a half-edge h until we encounter h again.

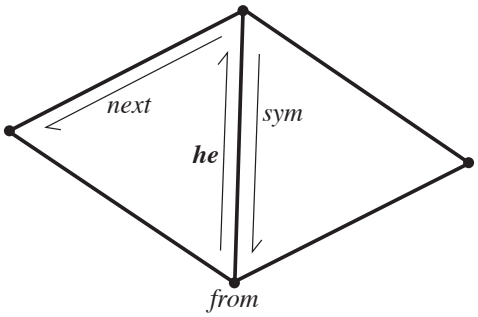

Fig. 7: *A single half-edge in a mesh.*

The *quad-edge* data structure is a related mesh representation which stores information about the *dual* of the original or *primal* mesh [23]. There is a bijection from faces in a primal mesh to vertices in its dual, and similarly a bijection from primal vertices to dual faces. Pairs of dual vertices which are mapped to from adjacent primal faces are adjacent (i.e., they are connected by a dual edge). The quad-edge data structure simultaneously represents primal and dual meshes by linking edges with their dual, symmetric, and dual symmetric edges. Implementations of both half-edge and quad-edge data structures are often limited to the representation of surfaces (i.e., non-manifold regions are not representable).

There is no real standard format for storing half- and quad-edge meshes on disk, but in general these two representations require just as many (if not more) indirect references as triangle soup in order to process local neighborhoods in the mesh.

2.3 Out-of-Core Processing

There are two main classes of out-of-core processes: *batched* processes, which stream data from disk through memory in a fixed order, and *online* processes, which can make data-dependent queries to disk. Due to the high latency of disk access, online processing is only successful if queries are *coarse*, i.e., large blocks of data are read in and used at once. Since streamed data has a fixed order, batched processes are almost always efficient, so long as the total number of passes through the data is small.

External sorting algorithms, which sort files on disk, are a common tool in OoC mesh processing. One common way to perform an external sort is to partition a file into segments which each occupy less than half of available memory, sort each segment using a fast in-core sort such as *quick sort*, and then *merge* the sorted segments two or more at a time using a merge operation like the one used in *merge sort* [50].

To cope with slow disk access, tree data structures used in OoC algorithms are typically much wider (i.e., higher branching factor) than their in-core counterparts. Rather than perform many reads which use only a small part of each disk block, the trees store disk block-sized nodes, and perform a search which is relatively inexpensive once nodes are loaded into memory. *B-trees* are a classic example of a general-purpose disk-based tree [3]. In-core tree data structures can be *externalized* by compressing subtrees with low branching factors (e.g., binary trees) into external nodes with higher branching factors (e.g., high-degree B-trees).

For a more extensive introduction to out-of-core processing relevant to computer graphics and visualization, see the IEEE Visualization 2002 course notes [48].

2.4 Simplification

Because OoC meshes are so large, it is not surprising that *simplification* algorithms are a popular topic in OoC processing. Therefore, we give a brief review of in-core simplification methods. Simplification

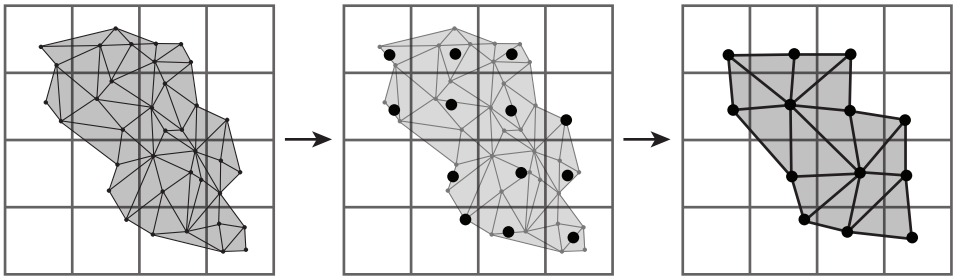

Fig. 8: *Using the fine input mesh (left), a representative vertex is chosen for each cell (center), and appropriate representative vertices are connected to form the coarse mesh (right).*

reduces the number of triangles used to represent a surface, while still retaining important geometric and topological characteristics.

2.4.1 Geometric Error

Simplification algorithms aim to minimize the difference between the original and simplified mesh at a given *resolution* or number of triangles. One measure of geometric error is the *Hausdorff distance*. If we define the *underlying space* $|S|$ of a mesh or simplicial surface S as the union of all of its 2-simplices, then the Hausdorff distance d_H between simplicial surfaces S_1 and S_2 can be written as

$$d_H(S_1, S_2) = max_{p \in S_1}\{min_{q \in S_2}\{d(p, q)\}\}$$

where $d : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ is the usual Euclidean distance. The Hausdorff distance yields the maximum distance of any point in S_1 from its nearest neighbor in S_2. However, this distance is not the best measure of the similarity between two surfaces. For instance, a large Hausdorff distance may either indicate that two surfaces are globally very dissimilar, or that only a single local region differs. Additionally, two surfaces which are identical up to isometry will still have a non-zero Hausdorff distance. Variations on Hausdorff distance account for these issues, but for the purposes of mesh simplification Hausdorff distance is not very useful because it takes a long time to compute (or even estimate). The "Metro" package is a popular tool for approximating Hausdorff distance via sampling [11].

A more efficient metric commonly used for mesh simplification is the *Quadric Error Metric* (QEM) [20], [19], which gives a local measure of error at each vertex of a mesh. In this context, a *quadric* is a function which maps a query point p to the sum of squared distances between p and each of a set of planes. However, the set of planes is purely conceptual, and in practice quadrics have a highly compact representation. An additional benefit is that a quadric which represents the distance from a disjoint union of sets of planes $P_1 \sqcup P_2$ can be computed by simply adding the coefficients used to represent the quadrics corresponding to P_1 and P_2. This feature makes quadrics extremely useful for mesh simplification, as discussed in sections 2.4.2 and 2.4.3.

2.4.2 Vertex Clustering

One major class of simplification algorithms is based on *vertex clustering* [45]. Vertex clustering partitions the ambient space of the surface (typically $\mathbb{R}^3$), and finds a single *representative vertex* for each element of the partition which contains vertices in the original mesh. Triangles in the simplified mesh are constructed from any three distinct representative vertices which correspond to vertices of a triangle in the original mesh.

A typical partition divides space into regular grid cells, as shown in figure 8. The spacing of the grid determines the resolution of the output mesh. Due to the regularity of this partition, vertex clustering using a regular grid often results in visible artifacts, especially on a coarse grid.

A number of schemes are used to determine the representative vertex for each partition. A common scheme is to put the representative at the average location of vertices within its cell, which effectively smooths the surface. Alternatively, a weight is assigned to each vertex based on, e.g., perceptual

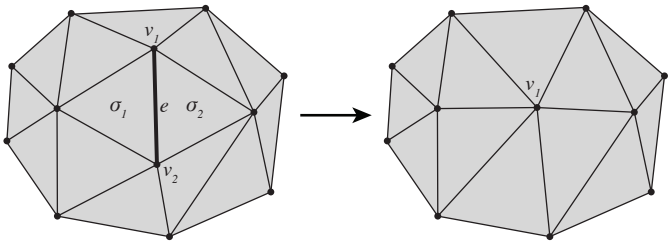

Fig. 9: *A single edge contraction eliminates the two triangles which contain e from the mesh.*

criteria, and the vertex of highest weight becomes the representative. More recent schemes [37] [38] compute a quadric Q for each cell equal to the sum of the quadrics of each vertex within the cell. The representative vertex is then placed at a location within the cell where Q is smallest.

2.4.3 Vertex Pair Contraction

Another class of mesh simplification algorithms uses *vertex pair contraction* as its basic operation [20], [19]. Vertex pair contraction is illustrated in figure 9: endpoint v_1 of edge e is repositioned, endpoint v_2 is removed, and triangles previously incident on v_2 are now incident on $e1$ instead. Triangles σ_1 and σ_2 become degenerate during this process and are removed from the mesh. The quadric Q associated with the contracted vertex is equal to the sum of the quadrics of the vertices in the pair. Simplification begins with a full-resolution mesh and contracts vertex pairs until the target resolution is reached.

As the name implies, either endpoints of an edge *or* an arbitrary pair of vertices may be contracted during a simplification step. Candidates for contraction are often determined by specifying a fixed distance within which vertices are paired. A single vertex pair contraction may not remove any faces from the mesh. Contracting vertex pairs which are not connected by an edge permits topological simplification of models with, e.g., many components or many small holes.

Pairs may be contracted in any order, but by far the most common procedure is to associate a quadric with each vertex, and contract the pair which increases the quadric error the least. To efficiently implement this procedure, the target vertex location for each pair contraction is computed, along with the associated quadric error. Vertex pairs are placed in a min heap, keyed by their associated error. Pairs are then removed from the top of the heap and contracted until the target resolution is reached, making necessary updates to neighbors' quadrics as each pair is contracted.

The location of a contracted vertex is determined using either *subset placement* or *optimal placement*. Subset placement simply uses the location of the vertex whose error is smaller under the combined quadric Q. Optimal placement tries to find the point which minimizes Q, reverting to subset placement if the solution is ill-defined.

Overall, this simplification scheme is greedy and not globally optimal. However, it is the most popular technique for in-core simplification due to its efficiency and the quality of its output. The method is also robust, handling non-manifold meshes. Figure 10 shows a model of hand at various levels of simplification. The surface is well-preserved at the first few levels of simplification, despite

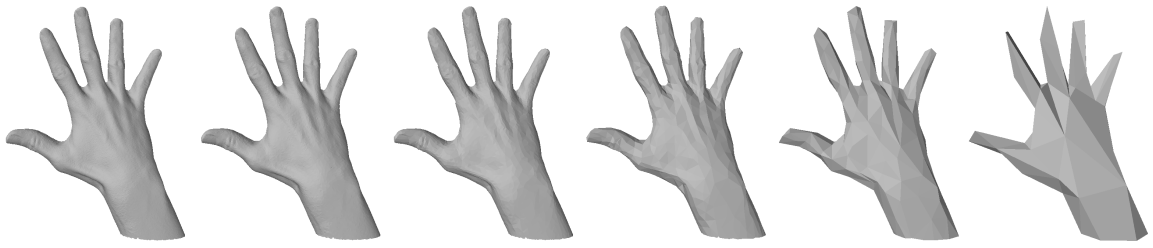

Fig. 10: *Hand model simplified using QSlim at 273k, 34k, 9k, 2k, 500, and 100 triangles (left to right).*

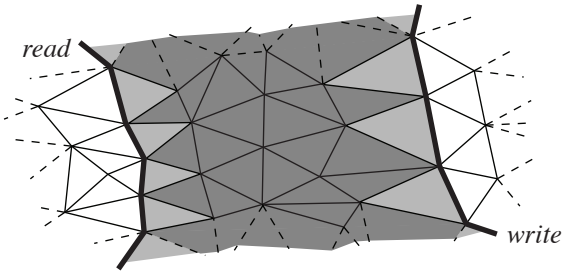

Fig. 11: *Required consistency with triangles on disk (white), makes triangles near* read *and* write *boundaries of a triangle stream (light gray) less malleable than other in-core triangles (dark gray).*

a drastic reduction in triangle count. Additionally, geometric features (fingers) are maintained at the lowest resolutions.

3 OoC Mesh Data Structures

This section examines the evolution of data structures used for OoC mesh processing. Many of these data structures were originally presented in the context of specific OoC algorithms, which are discussed separately in section 4.

A small number of high-level attributes help characterize each of these data structures. These attributes include

- how the mesh is partitioned on disk
- the order in which triangles are processed
- what type of query operations are supported

The first two items affect the efficiency of the representation, whereas the latter two affect the potential applications of the data structure. Triangle soup (section 3.1) partitions the data into single triangles, processes triangles in an arbitrary order, and only allows the mesh to be accessed as a stream. A multiresolution representation (section 3.2) partitions data into regular grid cells and permits queries of the surface based on spatial frequency, but is not designed for efficient traversal of the entire mesh. Ordered mesh data structures (section 3.3) represent the mesh as a spatially coherent sequence of triangles, but like triangle soup permit the mesh to be interpreted only as a stream.

3.1 OoC Triangle Soup

As mentioned in section 2.2.3, triangle soup is a convenient representation for meshes due to its simplicity. Therefore it is natural that early OoC algorithms attempted to use triangle soup as a mesh representation [37] [38] [53].

Wu and Kobbelt use a batch processing technique for performing simplification on triangle soup [53]. A generalization of their method describes how one might stream triangle soup for OoC processing: all operations are classified as *read*, *write*, or *process*, and each operation has an associated effect on memory. For instance, *read* will increase memory usage, while both *process* and *write* may reduce memory usage. The goal is to organize operations in the algorithm such that the available memory is well-utilized. Because disk operations are expensive, *read* and *write* operations are accumulated until a large block of contiguous data in the file is needed - only then is data read or written.

For this kind of batching scheme to succeed, *read* and *write* requests must correspond to coherent disk locations (and subsequently, *process* operations will only be able to operate on triangles from a small number of disk-coherent collection of triangles). Wu and Kobbelt argue that many processes which produce mesh data (e.g., range scanning and marching cubes isosurfacing [39]) are naturally

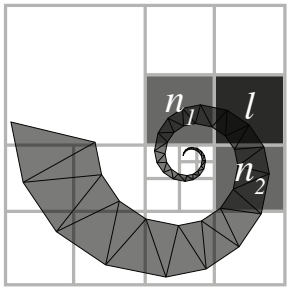

Fig. 12: *Loading a leaf l in an OEMM hierarchy also loads any leaves sharing a triangle with l (in this case, n_1 and n_2).*

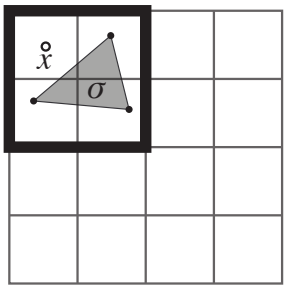

Fig. 13: *A triangle σ is stored in the marked node (bold) where it is represented by a single vertex x. In the next finest level (dark gray) it will no longer be degenerate since each vertex is in a different node.*

coherent, and otherwise pre-sorting triangles by one of their coordinates will suffice. (An illustration characterizing the desired stream order is given in figure 11 - note that triangles which share vertices with triangles not presently in memory cannot be modified as extensively as those with neighbors in memory.)

In reality, many original datasets are fairly coherent, as discussed by Isenburg and Lindstrom [29]. However, the suggestion to sort triangle data foreshadows better OoC data structures, as discussed in section 3.3.

3.2 Spatial Hierarchies

Spatial hierarchies are data structures which yield efficient access to geometric datasets by recursively subdividing space. Spatial data structures are useful for queries of high locality (e.g., ray-mesh intersection used in ray tracing) or of a particular frequency (e.g., MIP mapping used for texturing). Common in-core spatial hierarchies in visualization and graphics include BSP-trees [17] and bounding volume hierarchies [32]. Several OoC spatial hierarchies have also been developed, such as the *binary-blocked I/O interval tree*, used for isosurface extraction [9].

Cignoni et al present a version of an *octree* (a specific type of BSP-tree) adapted to OoC algorithms called an *Octree-based External Memory Mesh* (OEMM) [12]. A cube bounding an input mesh is recursively subdivided along each axis into eight equally-sized cells until all cells contain a similar number of vertices - these cells are the leaves of the octree. Each leaf is recorded as a separate entity on disk, storing all vertices and faces contained entirely within the cell. (Note that just as in a B-tree, nodes stored on disk are particularly wide.) For a face f with vertices in multiple cells, the cell c with the lowest lexicographic index stores f, and the vertices of f within other cells store the index of cell c. This scheme enforces consistency among cells by keeping only one copy of each vertex. However, it requires that any cells sharing a face with a leaf be loaded into memory in order to process that leaf.

An interesting feature of OEMMs is that the index of a vertex is assigned within a unique range of indices allocated to its containing leaf. By partitioning the range of 32-bit integers evenly among all the leaves, each leaf likely has many unused indices in its range. This allows the mesh topology to be locally modified by loading a leaf and its neighbors into an intermediate indexed mesh, processing the mesh, and then replacing the corresponding leaves in the octree, giving vertices an index in the appropriate range. Leaves which become either too sparse or too dense can be merged or split, maintaining a consistent node size (and thus making good use of each disk access).

A more recent approach [47] stores the original mesh at the finest level of an octree, along with a *multiresolution* representation of the mesh stored at coarser levels of the tree (i.e., a successively lower-resolution approximation is stored at each level toward the root). This multiresolution representation partitions space into a uniform grid of cells (which become the leaves of the tree), and progressively clusters leaves to build the tree. During the clustering process, each node keeps track of a representative quadric (along with other information), and triangles from the original mesh are stored in the finest level of the tree in which they are degenerate (see figure 13).

Unlike an OEMM, the multiresolution representation allows the mesh to be efficiently sampled at a variety of spatial frequencies. For instance, a set of regular samples spaced at the resolution of one leaf cell over the entire volume of the mesh would require that every leaf be loaded into memory if using an OEMM. Alternatively, a low frequency sample could be taken by extracting triangles from the appropriate level in the multiresolution hierarchy. Additionally, the triangles sampled from the OEMM would not yield an approximation of the original surface, as would the triangles extracted from the multiresolution tree. The success of a multiresolution hierarchy indicates something interesting about mesh data structures, namely that there is a large variety of sampling patterns we may want to apply to a geometric object. Other efficient data structures might be designed around *a priori* knowledge of sampling patterns for a particular class of algorithms.

In both of the octree-based methods, the hierarchy that references nodes on disk is stored entirely in-core. This requirement might appear to limit the size of a mesh stored in one of these formats, but in practice the memory used is not significant. For instance, an OEMM encoding a 10 billion triangle mesh requires only 40 MB of memory for the hierarchy.

A more significant problem with these data structures is that they do not address traversal order either within a node or over the entire hierarchy. For instance, one proposed application of OEMMs traverses octree nodes in lexicographic order. Due to faces shared among leaves, each leaf will be loaded into memory multiple times during traversal, yet only a small piece of each leaf will be processed when loaded in as a neighbor (i.e., only the shared faces). This behavior directly contradicts our requirement for OoC trees that a large fraction of the data in each block be heavily utilized. Additionally, no suggestion is given on processing order for triangles within a node. Although this is not a strictly OoC issue, it will affect performance in finer levels of the cache hierarchy.

3.3 Ordered Mesh Data Structures

3.3.1 Out-of-Core Meshes

As in the hierarchical schemes, the *Out-of-Core Mesh* (OoCM) of Isenburg and Gumhold partitions the mesh into small segments which can be loaded into memory [27]. Once an OoCM has been built, triangles can be efficiently organized into an sequence such that streaming through the data in order maintains a local neighborhood on the surface.

Since segments do not need to be organized hierarchically in an OoCM, the scheme used to partition the mesh can be more flexible. Isenburg and Gumhold partition the mesh into connected clusters of triangles in the following way. First, a regular grid is imposed on the mesh, and the centroid of vertices contained within each cell is computed. Next, a graph is constructed where each vertex corresponds to a cell, and edges are determined by finding the k-nearest neighbors of each cell with respect to their associated centroids. Graph vertices are weighted according to the number of vertices in the corresponding cell. Finally, a partition of the graph is computed such that total vertex weight is fairly consistent among elements of the partition. The OoCM is based around a half-edge data structure, so vertices and half-edges contained entirely inside a cluster are stored together on disk. Half-edges which cross a boundary between clusters are stored in the cluster which contains their *from* vertex.

Although the clustering process yields segments which all occupy a similar amount of disk space, it may not be optimal in terms of, e.g., the boundary length of each cluster. Clusters with a short boundary are desirable because they contain fewer triangles which are dependent on data in neighboring clusters. During stages of processing which do not modify the mesh, some dependencies can be avoided by storing a copy of any half edge which crosses a boundary with the corresponding clusters.

Used by itself, the OoCM is not a particularly interesting OoC data structure. However, the partitioning used to generate an OoCM is more closely related to the *surface* of the mesh than in previous methods, which strictly partition the ambient space. A topological partition makes it easier to develop streaming mesh representations which preserve surface locality, as discussed in sections 3.3.3 and 3.3.4.

3.3.2 Rendering Sequences

Rendering sequences are a common paradigm for efficient rendering of *in*-core meshes [13]. *Triangle*

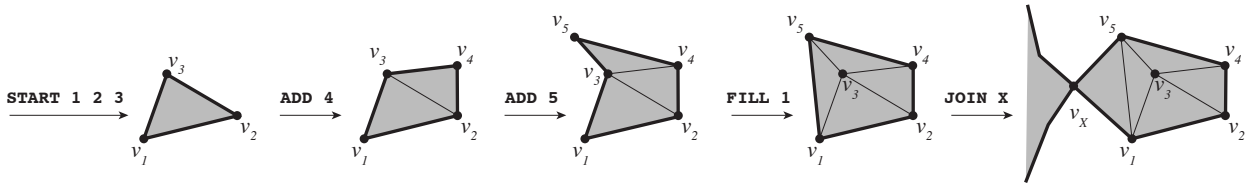

Fig. 14: *A series of processing sequence operations constructs a mesh.*

strips are a specialization of rendering sequences which are supported by graphics cards in order to reduce vertex processing.

The most common implementation of triangle strips (as is seen in, e.g., OpenGL) maps a sequence of vertices $\langle v_i \rangle$ to a set of triangles. The first three vertices in the sequence construct a triangle, and any subsequent vertex v_n creates a triangle along with vertices v_{n-1} and v_{n-2}. This procedure reduces redundant vertex processing, since vertices of adjacent triangles are shared. Triangle stripping is an attractive scheme for graphics hardware, since it requires only a small FIFO *vertex cache*. *Generalized triangle stripping* allows a new vertex to selectively replace either v_{n-1} or v_{n-2} in the vertex cache, permitting a greater range of vertex orderings.

Rendering sequences provide a framework for further generalizations of triangle strips (called *triangle meshes*) by providing a *mesh buffer* or queue of cached vertices somewhat larger than the two-vertex cache used by ordinary triangle strips. A rendering sequence can be thought of as an "instruction set" for triangle strips, allowing vertices to be accessed from and added to this queue.

Finding a good decomposition of a mesh into triangle strips or triangle meshes is fairly challenging. Breaking a mesh up into generalized triangle strips is equivalent to finding a *Hamiltonian path decomposition* of the edges of the dual mesh [15] [2]. Optimal decomposition into a triangle meshes is also difficult [5].

In some sense, triangle strips are a solution to OoC representation, since data dependence for each triangle is extremely local. For many applications, however, triangle strips do not yield enough topological information to be useful. As discussed in the next section, larger vertex caches will permit similar but more robust solutions.

3.3.3 Processing Sequences

Just as rendering sequences provide a framework for mesh representations subject to a small vertex cache, *processing sequences* provide a framework for representations subject to limited memory [28].

A processing sequence consists of an ordered list of five basic operations: *start*, *add*, *fill*, *join*, and *end* . The first operation in a processing sequence is the *start* operation, which adds a new triangle to the mesh. The *add* operation specifies a single vertex, which constructs a triangle with two border vertices of the current mesh. The *fill* operation connects two border vertices to form a triangle. The *join* operation creates a triangle with two adjacent vertices from one border and one vertex from another. Finally, the *end* operation creates a triangle from three edges already on the border, filling a hole in the mesh.

```
v 0.213 0.233 0.129 4
v 1.870 0.721 0.224 3
v 1.684 1.048 0.387 4
START 1 2 3
v 0.374 2.191 0.473 3
ADD 4
v 0.261 0.746 1.854 4
ADD 5
FILL 1
JOIN X
```

Fig. 15: *Mock-up of a processing sequence for the geometry in figure 14.*

A processing sequence also contains each mesh vertex at some point before it is referenced by one of these operations. In order to *finalize* vertices, i.e., evict from them memory once they are no longer needed, some additional information needs to be stored in the sequence. This information may indicate the last time a vertex is used, or specify the number of times it is referenced (corresponding to the *valence* of the vertex).

Figure 14 illustrates a mesh being constructed using most of the sequence operations (the vertex labeled X represents a vertex loaded into memory prior to the beginning of the example sequence). Although processing sequences are independent of a specific file format, a mock-up of a human-readable

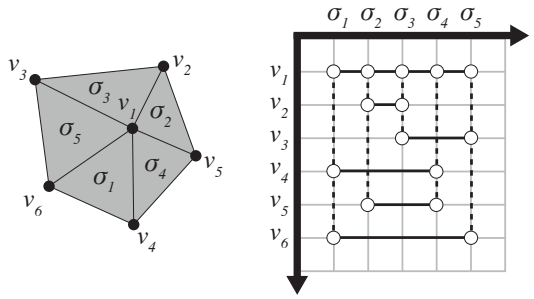

Fig. 16: *Meshes ordered arbitrarily yield fragmented layouts with large triangle and vertex spans.*

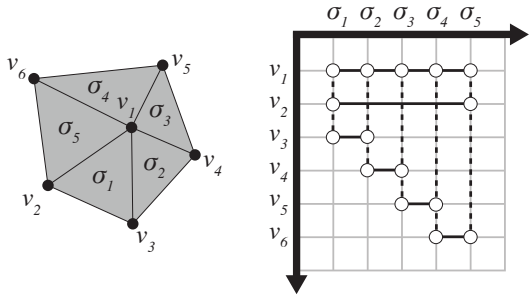

Fig. 17: *Carefully ordered meshes yield diagonally dominant layouts with small triangle and vertex spans.*

format corresponding to figure 14 is shown in figure 15 (note that vertex locations do not correspond to those shown in the illustration). The integer value associated with each vertex specifies its valence.

Processing sequences are able to reference a much larger set of vertices than are rendering sequences. The size of this set is determined by the memory resources allocated to in-core processing. However, a processing sequence must respect this memory limitation, because there is no mechanism for reintroducing a vertex once it has been evicted. An additional feature of processing sequences is that they can be streamed out in exactly the same manner as they are streamed in. This not only allows streaming output, but also permits pipelined processes where the result of one stream is fed immediately into the input of another.

Just as with rendering sequences, we need some way to construct a processing sequence from an unorganized mesh. One way to determine a processing sequence is to find a *spanning tree* of the edges of the dual mesh, and traverse this tree in a coherent order. This problem is examined extensively within the mesh compression literature [43] [44] [49] - in fact, Isenburg and Gumhold present a scheme for building a processing sequence derived from compression [27]. However, the depth-first traversal of the tree used by Isenburg and Gumhold turns out to be sub-optimal, as discussed in the next section.

3.3.4 Streaming Meshes

Isenburg and Lindstrom introduce a useful visualization of coherence for indexed meshes called a *layout diagram* along with several related metrics, which they use to develop the *streaming mesh* data structure. Figures 16 and 17 show two meshes and their corresponding layouts. Each column in a layout corresponds to a face, and each row corresponds to a vertex. Vertices of a face are marked with a white circle, and connected by a dashed vertical line. The length of this line is referred to as the *vertex span* of a triangle, indicating the coherence of vertices referenced by the triangle. Solid horizontal lines drawn between the first and last instance of a vertex indicate the *triangle span* of a vertex, indicating, e.g., how long a vertex must be kept in memory before it can be finalized. Layout diagrams yield useful visual interpretations of mesh organization - for instance, the maximum number of overlapping triangle spans (called the *vertex width*) gives a lower bound on the maximum number of vertices which need to be concurrently loaded into memory to reconstruct an indexed mesh from a stream. Diagonally dominant layouts represent high overall coherence, since both vertex span and triangle span are small.

Layout diagrams do not fully characterize the coherence of an indexed mesh unless we put further constraints on the order of data in the corresponding file. For instance, even though every vertex of a mesh may be referenced by a coherent sequence of triangles, the vertices and faces might be stored in two separate chunks (as in the OBJ file from figure 6). To prevent this kind of arrangement, streaming meshes interleave vertex and face information, making the stream more *compact*. A stream in which every vertex is referenced by the closest triangle in the stream is called *vertex-compact*, and a stream in which every face references the closest vertex in the stream is called *triangle-compact*. (Additionally, streaming meshes contain finalization information so that each vertex needs to be loaded into memory only once, just as with processing sequences.)

Once a mesh has been compacted, its mesh layout provides a better picture of actual stream coherence. For instance, the vertex width of a vertex-compact stream directly indicates the amount of memory needed for vertex storage. Any mesh can be expressed as a vertex-compact stream. For example, one could always write a triangle immediately after all of its vertices have been specified.

The real purpose of mesh compaction, of course, is to increase coherence and reduce the amount of memory required to stream through an OoC mesh. However, not all streams can achieve good coherence through compaction alone. For instance, the labeling of vertices and faces in figure 16 precludes any stream ordering better than the one shown in the corresponding mesh layout. By relabeling the mesh we get a layout in which the majority of vertices only need to be loaded into memory for a short period of time (figure 17).

Isenburg and Lindstrom propose three schemes for re-ordering mesh layouts: *spatial sorting*, *topological sorting*, and sorting based on *spectral sequencing*. Spatial sorting simply re-orders vertices with respect to their projection onto some canonical direction (e.g., vertices might be sorted by their x-coordinate). Corresponding triangles are then re-indexed and placed compactly in the stream. Topological sorting sorts vertices with respect to relative proximity on the underlying surface. Isenburg and Lindstrom perform either a breadth-first search on vertices or a depth-first search on triangles, though other topological methods could be applied. Not surprisingly, a depth-first traversal tends to produce vertices with a higher span than does a breadth-first traversal, since it may take a long time to finalize vertices from the beginning of the search. Spectral sequencing first finds the eigenvector corresponding to the second-smallest eigenvalue of the Laplacian matrix of the mesh. This eigenvector, called the *Fiedler vector*, defines a function over the vertices which has one minimum and one maximum. Vertices are then ordered by starting at the minimum and proceeding toward the maximum (or vice versa).

All of the proposed methods are heuristic, but in practice result in extremely coherent layouts. In reality, an optimal layout is not really required: any layout which will fit into memory will do. However, good reordering will benefit processing in finer levels of the cache hierarchy. While spatial or topological sorting produce good layouts for many meshes, meshes with a winding or fractal structure may benefit from (more expensive) spectral sequencing.

Although streaming meshes provide a highly efficient mesh representation for applications which can operate on the mesh as a stream, they do not provide any benefit for other types of access patterns, as discussed in section 3.2.

3.4 Alternative Representations

Although meshes are the most popular representation for in-core geometry processing, it is worthwhile to investigate alternative surface representations which may yield better interaction with large datasets.

3.4.1 Point-based Representations

Recently, *point-based* representations have become a popular topic in surface representation [36] [33]. Point based models consist of only point samples from a surface, possibly including information about normals, colors, etc. A collection of points unambiguously defines a surface only if certain sampling criteria are satisfied [1]. Although point-based models do not explicitly define connectivity, a surprising number of surface operations are still possible [41], usually by assuming connectivity among nearby points.

Point-based methods have long been used to visualize large data sets, most notably the *QSplat* rendering system used by the Digital Michaelangelo Project [46]. The QSplat system builds a multiresolution bounding volume hierarchy of points, attaining speeds suitable for interactive visualization. In general, point based visualization of large models is particularly effective because downsampling can be performed by taking a subset of the points without worrying about connectivity. Although points in a data set do not reference each-other explicitly, it may be useful to give points an ordering similar to those used for streaming meshes in order to efficiently estimate connectivity.

3.4.2 Geometry Images

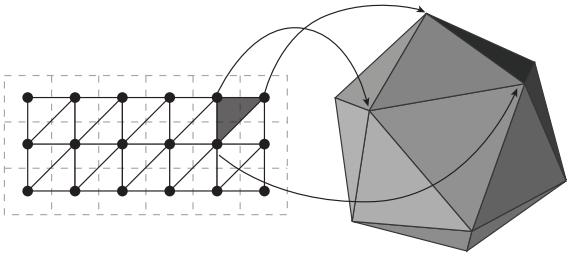

Fig. 18: *Pixels of a geometry image (left, dashed lines) implicitly encode the connectivity of a triangle mesh (left, solid lines).*

Geometry Images are a surface representation introduced by Gu et al [22] which cut a surface into a topological disk so that it can be homeomorphically mapped to a rectangular image. The x, y, and z coordinates of points on the surface are then assigned to the red, green, and blue components (respectively) of corresponding pixels. A triangle mesh can be reconstructed from this representation by interpreting each pixel as a vertex, and connecting vertices which correspond to neighboring pixels into a regular grid of triangles (figure 18).

Geometry images may provide a fairly robust representation for OoC processing: not only is the data layout highly coherent, but the connectivity of the mesh is uniform. An OoC mesh could be efficiently processed by accessing one "tile" or rectangular segment of the image at a time, with a small amount of additional work along tile boundaries (a.k.a., *topological sideband*). Additionally, a multiresolution representation could be constructed by downsampling the geometry image, creating a MIP map-like pyramid of images. This structure would permit sampling of the surface at various frequencies, similar to the hierarchical representation discussed in section 3.2.

There are several difficulties involved in using geometry images as an OoC representation. First, the original mesh needs to be cut into a topological disk and parameterized. Many cutting and parameterization algorithms require global information about a surface (making OoC processing difficult), though there is no evidence that these tasks are intractable. Second, conversion to a geometry image effectively remeshes the surface, giving it different connectivity. For most OoC applications a change in mesh topology is not a problem, however, as evidenced by the simplificaton frequently applied to large datasets. Finally, parameterization of a non-developable surface to a rectangle incurs a large amount of parametric distortion. In order to preserve the fidelity of the original surface, geometry images representing more triangles than in the original (already massive) mesh are likely required. However, distortion could be reduced by converting each element of a partition of the mesh into its own geometry image [7].

3.4.3 Volumetric Representations

Volumetric data consists of a subdivision of three dimensional space where each cell takes a sample value from a three dimensional function. Many OoC surface datasets originate as isosurfaces of large volumetric datasets produced by simulation or medical imaging. A naive volumetric representation subdivides space into regular grid cells. As with geometry images, one advantage of a regular volumetric representation is that data layout and connectivity is highly regular, making it easy to segment into chunks suitable for in-core processing. However, regular volume data is generally wasteful and asymptotically more expensive to process than a mesh due to the increase in dimension: the dimensionality of most scanned surfaces is much closer to two than to three, although isosurfaces extracted from simulation can be more complex.

More interesting volume representations are irregular or sparse (or both). The *meta cell* data structure partitions an irregular tetrahedral mesh into chunks which fit in memory [9]. Similar to the clusters used by an OoCM, vertex data within each chunk is referenced by a local index, preventing data redudancy among tetrahedra and permitting simple processing within a meta-cell. Although a volume

isosurface may seem like a wasteful way to represent a surface, some types of OoC processing benefit from a volumetric representation, as mentioned in section 4.2.2.

4 OoC Mesh Algorithms

Although a large variety of OoC data structures have been developed (3), it is not always obvious how to restructure algorithms to work with these new structures. Because OoC data structures were designed to access meshes with high spatial locality, so too must algorithms be rephrased. Fortunately, many common geometry processing tasks such as surface simplification, smoothing, and visualization are naturally local. On the other hand, more global operations such as surface parameterization remain challenging.

4.1 OoC Simplification

As noted before, simplification is an essential OoC algorithm, since most applications cannot process such large datasets (nor do they need to). Unlike other areas of signal processing, high-fidelity mesh data is often downsampled *before* processing takes place, mainly due to the huge discrepancy between source and target resolution. One might question the merit of acquiring such large datasets if only to simplify them. However, there are good arguments for maintaining such massive datasets, such as the desire to archive historical artifacts.

Early approaches to OoC simplification were based exclusively on vertex clustering (section 2.4.2) as the simplification operation and used OoC triangle soup as the mesh representation [37] [38]. The *Out-of-Core Simplification* (OoCS) method subdivides the region containing the input mesh with a regular grid, and cells accumulate quadric information as triangles are streamed in from disk. Because the stream of triangles is unordered, there is no way to know when specific grid cells will be needed. Rather than swap grid cells out to disk and risk severe thrashing, the entire grid is stored in memory. Although the grid is given a sparse representation (using a hash table), the potential size of the output mesh is still limited by available memory resources. A modified version of the method, called *OoCSx*, lifts this restriction by streaming computed quadrics to disk, along with the index of the corresponding cell. Once quadrics have been computed for all triangles, the quadric file is sorted using an external sort (section 2.3), and the sorted quadric file is traversed linearly to compute the final quadric for each cell. Although OoCSx requires less memory than OoCS, it runs approximately two to four times slower due to the expensive external sort.

As noted before, simplifications produced via vertex clustering are typically inferior in quality to those produced via vertex pair contraction 2.4.3. However, vertex clustering is a more efficient process for large models because it eliminates many faces at once, whereas contraction reduces complexity one pair at a time. To reconcile the discrepancy between these two methods, Garland and Shaffer developed a *multiphase* approach [21], which performs an initial simplification out of core using vertex clustering, followed by a higher-quality in-core simplification using vertex pair contraction. The key observation is that the quality of the latter simplification process is significantly improved by maintaining vertices'

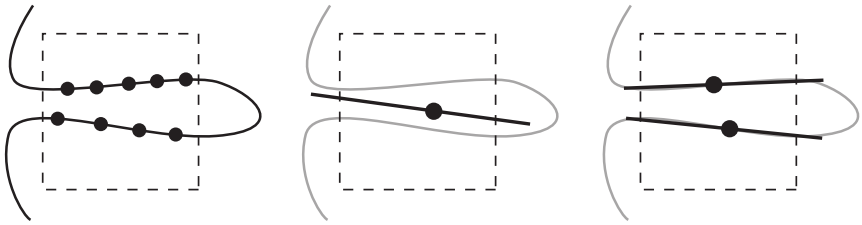

Fig. 19: *Standard vertex clustering indiscriminately clusters all vertices in an input cell (left), "pinching" the surface together (center), whereas topological traversal helps preserve distinct features by maintaining a count of active vertices (right).*

quadrics from the earlier simplification, rather than computing quadrics directly from the simplified mesh.

In order to achieve better performance while still using a fixed amount of memory for both input and output meshes, Wu and Kobbelt describe a streaming algorithm for simplification based on vertex pair contraction [53]. As mentioned earlier, Wu and Kobbelt assume that the stream of polygons generated for most meshes will preserve locality enough to process a coherent *front* of triangles. Since the entire mesh is too large to be loaded into memory, they cannot construct the usual global min heap of quadrics used in pair-contraction simplification. However, they make the observation that simplifying local regions of the mesh to the desired percentage of the input resolution yields almost identical results to globally picking the pair with the smallest quadric (i.e., the particular order of simplification is not significant). Therefore, the algorithm proceeds by probabilistically either reading, writing, or decimating geometry, with probabilities picked such that the target resolution is reached.

Streaming simplification can be performed in a similar manner within the framework of processing sequences [28]. Isenburg et al process vertex pairs which fall between the *read* and *write* boundaries of the sequence in a manner almost identical to Wu and Kobbelt. They also describe a streaming version of vertex clustering similar to OoCS, but are able to write and deallocate grid cells which are no longer interacting with the processing boundary. Processing triangles in topological order also prevents "pinching" artifacts commonly seen in vertex clustering, since cells will reactivate and write new triangles should they be encountered by the processing boundary again (figure 19).

The same observation (i.e., that a particular simplification order is not important) enabled Cignoni et al to apply the OEMM data structure to OoC simplification [12]. The algorithm simply traverses the octree in lexicographic order, simplifying each node locally. Since each leaf which shares faces with the current leaf is also loaded into memory, simplification is correctly applied at leaf node boundaries.

Shaffer and Garland also use an octree for simplification, but take a different approach using their multiresolution representation [47]. Locally, a simplified version of the surface already exists within a multiresolution hierarchy. By simply specifying the appropriate *cut* through the octree and extracting the corresponding triangles, a simplified mesh is generated. Shaffer and Garland permit the user to either specify a maximum quadric error or a target number of triangles in order to determine the amount of simplification. The cut is determined from one of these parameters by performing a breadth first search from the root of the tree, and adding triangles only if they exceed the error threshold (if a maximum error was specified), or if they contribute the maximum error out of all triangles on the traversal boundary (if a number of triangles was specified).

4.2 OoC Visualization

4.2.1 View-dependent Visualization

All of the OoC data structures in some way support rendering of large meshes. For instance, a mesh represented as triangle soup can be streamed off of disk while triangles are accumulated in a frame buffer. However, this process is highly inefficient, since many unseen triangles are drawn, and triangles are often smaller than the size of a single pixel. Therefore, several OoC visualization methods have been developed which adapt to viewing conditions.

The multiresolution representation of Shaffer and Garland [47] can be used for view-dependent rendering by specifying a cut through the octree based on viewing criteria (this procedure is similar to the multiresolution simplification described in section 4.1). In order to determine this cut, other surface information (e.g., normals) is accumulated in cells during octree construction.

The *Adaptive TetraPuzzles* method of Cignoni et al also uses a multiresolution hierarchy for OoC view-dependent rendering [10]. This method uses a tetrahedral decomposition of space rather than an octree, and constructs batches of triangles well-suited to rendering on a GPU.

More generally, precomputed visibility information can help determine what subset of geometry is needed for visualization [14]. For instance, if a model can be partitioned into a set of *cells*, then only the geometry visible from within that cell, called the *potentially visible set* (computed as a preprocess) needs to be loaded into memory. This kind of partitioning makes sense for, e.g., architectural models, but is less applicable to, e.g., large scanned models. Additionally, automatic partitioning and PVS

computation is a fairly difficult problem [34].

4.2.2 Ray Tracing and Global Illumination

Naive rasterization of a mesh is a linear-time operation, since at very least every triangle must be submitted to some kind of visibility test. For OoC models, ray tracing using a spatial acceleration structure presents an attractive alternative, because visibility tests are performed in $O(\log N)$ time with respect to the number of triangles in the scene. Rendering via point sampling also seems like an appropriate solution, since projected triangles from large datasets will often project to an area of less than one pixel on the screen. However, performing a ray intersection with a mesh traditionally requires that the entire mesh be loaded into memory and stored in a spatial hierarchy, so alternative schemes must be developed.

Pharr et al partition scene geometry using a regular grid, and store geometry from each cell in a contiguous region on disk [42]. Rather than intersecting rays with the scene in the order which they are generated, rays are enqueued and carefully reordered to avoid expensive cache misses. Rays can be re-ordered with fine granularity because the contribution of a ray traced path can be broken up into the contributions from each of its edges.

Fradin et al present a method for photon mapping large meshes, using a *cell-and-portal* decomposition similar to PVS [16]. A photon map is constructed for one in-core cell at a time, and photons which traverse cell boundaries wait in a queue until the appropriate cell is processed. However, just as with PVS, the cell-and-portal decomposition applies only to a limited class of objects (in this case, buildings).

Wald et al use an *implicit* kd-tree acceleration structure to ray trace isosurfaces of OoC volumetric data sets, exceeding the performance of rasterizing an equivalent polygonal isosurface using graphics hardware [51].

4.3 Other Algorithms

An assortment of other mesh processing algorithms have been modified to work with OoC representations. Isenburg and Gumhold have developed a mesh compression scheme which uses the same stream ordering as is discussed in section 3.3.1 [27]. A locality-preserving stream order aids compression, because more mesh data can be encoded as small differences along the stream. Shaffer and Garland take advantage of the same multiresolution representation used for OoC simplification and rendering to perform collision detection among gigantic meshes [47]. Isenburg et al use an ordered stream of vertices to compute the Delaunay triangulation of massive point sets [31]. Wood et al present an OoC method for removing spurious topological handles from large isosurfaces [52].

5 Conclusion

Recent development of efficient data structures and algorithms for digital geometry processing has been driven by the need to work with datasets which do not fit into a particular level of the cache hierarchy, namely main memory. In the near future, however, memory resources may cease to be the bottleneck, due to dwindling RAM prices and the arrival of consumer 64-bit architectures. Additionally, for certain types of data we may have already reached an upper limit on complexity: models added to the Stanford Scanning Repository in recent years are in fact *less* complex than those acquired during the Digital Michaelangelo Project. However, the ubiquity of vast memory resources in no way diminishes the value of techniques developed for OoC processing, due to the recent appearance of several highly-related problems.

One relevant issue is the decrease in single-processor speed, which has necessitated the shift to multi-CPU architectures. In order to effectively parallelize geometry processing algorithms, data must be broken into and processed in coherent chunks - a problem identical to OoC processing. Although the granulartiy of decomposition will be finer in parallel algorithms than in OoC ones, the same basic

principles apply. Perhaps the main difference is that the data can no longer be treated purely as a stream: effective load balancing will require better knowledge of the size and structure of the mesh.

Ideas from OoC processing will also be useful in developing geometry processing on *stream architectures* such as GPUs. Although vertex streams on current GPUs are topologically uninteresting (i.e., only simple triangle strips), next generation (DX10) hardware introduces *geometry shaders*, which enable access to a larger topological neighborhood around each primitive. Because the GPU internally references vertices in a format similar to indexed meshes, the same kind of reordering applied to streaming meshes should have a significant impact on the performance of these new shaders. Architectures such as IBM's Cell processor will have to simultaneously cope with parallelism (multiple cores *and* multiple SPEs!) and with streaming (each individual SPE).

Finally, in-core processing of meshes can benefit from intelligent stream re-ordering. Memory bandwidth is not increasing at the same rate as memory capacity, meaning RAM may eventually behave much like a disk drive relative to the overall cache hierarchy.

Improving the locality of geometric data has proven to be a simple and effective way to substantially enhance interaction with digital surfaces. The cost associated with data optimization is easily amortized, especially since the ratio of geometry created to geometry consumed is presently very small. The remaining obstacle to ubiquitous efficient mesh processing is support of efficient data structures by content providers and application developers, although the streaming meshes API provided by Isenburg and Lindstrom may help in this respect [30].

References

[1] Nina Amenta and Marshall W. Bern. Surface reconstruction by voronoi filtering. In *Symposium on Computational Geometry*, pages 39–48, 1998.

[2] Esther M. Arkin, Martin Held, Joseph S. B. Mitchell, and Steven Skiena. Hamiltonian triangulations for fast rendering. *The Visual Computer*, 12(9):429–444, 1996.

[3] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[4] Ethan D. Bloch. *A first course in geometric topology and differential geometry*. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

[5] Alexander Bogomjakov and Craig Gotsman. Universal rendering sequences for transparent vertex caching of progressive meshes. In B. Watson and J. W. Buchanan, editors, *Proceedings of Graphics Interface 2001*, pages 81–90, 2001.

[6] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Directed edges — A scalable representation for triangle meshes. *Journal of Graphics Tools: JGT*, 3(4):1–12, 1998.

[7] Nathan Carr, Jared Hoberock, Keenan Crane, and John Hart. Rectangular multi-chart geometry images. *Symposium on Geometry Processing*, 2006.

[8] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Symposium on Discrete Algorithms*, pages 139–149, 1995.

[9] Yi-Jen Chiang, Cláudio T. Silva, and William J. Schröeder. Interactive out-of-core isosurface extraction. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 167–174, 1998.

[10] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Adaptive tetrapuzzles — efficient out-of-core construction and visualization of gigantic polygonal models. In *Proc. SIGGRAPH 2004*, volume 23 of *ACM Transactions on Graphics*, 2004.

[11] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

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

[13] Michael Deering. Geometry compression. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 13–20, New York, NY, USA, 1995. ACM Press.

[14] Frédo Durand. *3D Visibility: analytical study and applications*. PhD thesis, Université Joseph Fourier, Grenoble I, July 1999. http://www-imagis.imag.fr.

[15] Francine Evans, Steven S. Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 319–326, 1996.

[16] David Fradin, Daniel Meneveaux, and Sebastien Horna. Out of core photon-mapping for large buildings. In *Rendering Techniques*, pages 65–72, 2005.

[17] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, New York, NY, USA, 1980. ACM Press.

[18] M. Garland. Multiresolution modeling: Survey & future opportunities, 1999.

[19] Michael Garland. Quadric-based polygonal surface simplification, 1998.

[20] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[21] Michael Garland and Eric Shaffer. A multiphase approach to efficient surface simplification. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 117–124, Washington, DC, USA, 2002. IEEE Computer Society.

[22] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 355–361, New York, NY, USA, 2002. ACM Press.

[23] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph.*, 4(2):74–123, 1985.

[24] Allen Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

[25] Paul S. Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. Technical report, to appear.

[26] Anil Hirani. Discrete exterior calculus, 2003.

[27] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes, 2003.

[28] M. Isenburg, S. Gumhold, and Jack Snoeyink. Processing sequences: A new paradigm for out-of-core processing on large meshes, 2003.

[29] M. Isenburg and P. Lindstrom. Streaming meshes. In *Proceedings of IEEE Visualization 2005*, pages 231–238, 2005.

[30] Martin Isenburg and Peter Lindstrom, 2005.

[31] Martin Isenburg, Yuanxin Liu, Jonathan Shewchuk, and Jack Snoeyink. Streaming computation of delaunay triangulations. In *Proceedings of SIGGRAPH 2006.*, 2006.

[32] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1986. ACM Press.

[33] L. Kobbelt and M. Botsch. A survey of point-based techniques in computer graphics. In *Computers & Graphics*, volume 28, pages 801–814, 2004.

[34] Sylvain Lefebvre and Samuel Hornus. Automatic cell-and-portal decomposition. Technical report, 2003.

[35] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3d scanning of large statues. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 131–144, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[36] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical report, University of North Carolina at Chapel Hill, 1985.

[37] Peter Lindstrom. Out-of-core simplification of large polygonal models. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 259–262, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[38] Peter Lindstrom and Cláudio T. Silva. A memory insensitive technique for large model simplification. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 121–126, Washington, DC, USA, 2001. IEEE Computer Society.

[39] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[40] M. Meyer, M. Desbrun, P. Schröder, and A. Barr. Discrete differential geometry operators for triangulated 2-manifolds, 2002.

[41] Mark Pauly, Richard Keiser, Leif P. Kobbelt, and Markus Gross. Shape modeling with point-sampled geometry. *ACM Trans. Graph.*, 22(3):641–650, 2003.

[42] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[43] J. Rossignac, A. Safanova, and A. Szymczak. 3d compression made simple: Edgebreaker on a corner table. In *Proceedings of Shape Modeling International*, 2001.

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

[45] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. *Geometric Modeling in Computer Graphics*, pages 455–465, 1993.

[46] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[47] Eric Shaffer and Michael Garland. A multiresolution representation for massive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 11(2):139–148, 2005.

[48] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics, 2002.

[49] Gabriel Taubin and Jarek Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

[50] Jeffrey Scott Vitter. External memory algorithms and data structures. In James Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms and Visualization*, pages 1–38. American Mathematical Society Press, Providence, RI, 1999.

[51] Ingo Wald, Heiko Friedrich, Gerd Marmitt, and Hans-Peter Seidel. Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572, 2005. Member-Philipp Slusallek.

[52] Zoe Wood, Hugues Hoppe, Mathieu Desbrun, and Peter Schröeder. Removing excess topology from isosurfaces. *ACM Transactions on Graphics*, 23(2), 2004.

[53] J. Wu and L. Kobbelt. A stream algorithm for the decimation of massive meshes, 2003.