Cache-Oblivious Mesh Layouts

Sung-Eui Yoon[1] Peter Lindstrom[2] Valerio Pascucci[2] Dinesh Manocha[1]

[1]University of North Carolina at Chapel Hill [2]Lawrence Livermore National Laboratory
http://gamma.cs.unc.edu/COL

Abstract

We present a novel method for computing cache-oblivious layouts of large meshes that improve the performance of interactive visualization and geometric processing algorithms. Given that the mesh is accessed in a reasonably coherent manner, we assume no particular data access patterns or cache parameters of the memory hierarchy involved in the computation. Furthermore, our formulation extends directly to computing layouts of multi-resolution and bounding volume hierarchies of large meshes.

We develop a simple and practical cache-oblivious metric for estimating cache misses. Computing a coherent mesh layout is reduced to a combinatorial optimization problem. We designed and implemented an out-of-core multilevel minimization algorithm and tested its performance on unstructured meshes composed of tens to hundreds of millions of triangles. Our layouts can significantly reduce the number of cache misses. We have observed $2-20$ times speedups in view-dependent rendering, collision detection, and isocontour extraction without any modification of the algorithms or runtime applications.

1 Introduction

Over the last few years, advances in model acquisition, computer-aided design, and simulation technologies have resulted in massive databases of complex geometric models. Meshes composed of tens or hundreds of millions of triangles are frequently used to represent CAD environments, terrains, isosurfaces, and scanned models.

Efficient algorithms for processing large meshes utilize the computational power of CPUs and GPUs for interactive visualization and geometric applications. A major computing trend over the last few decades has been the widening gap between processor speed and main memory speed. As a result, system architectures increasingly use caches and memory hierarchies to avoid memory latency. The access times of different levels of a memory hierarchy typically vary by orders of magnitude. In some cases, the running time of a program is as much a function of its cache access pattern and efficiency as it is of operation count [Frigo et al. 1999; Sen et al. 2002].

Our goal is to design cache efficient algorithms to process large meshes. The two standard techniques to reduce cache misses are:

1. **Computation Reordering**: Reorder the computation to improve program locality. This is performed using compiler optimizations or application specific hand-tuning.
2. **Data Layout Optimization:** Compute a cache-coherent layout of the data in memory according to the access pattern.

In this paper, we focus on data layout optimization of large meshes to improve cache coherence. A triangle mesh is represented by linear sequences of vertices and triangles. Therefore, the problem becomes one of computing a cache efficient layout of the vertices and triangles.

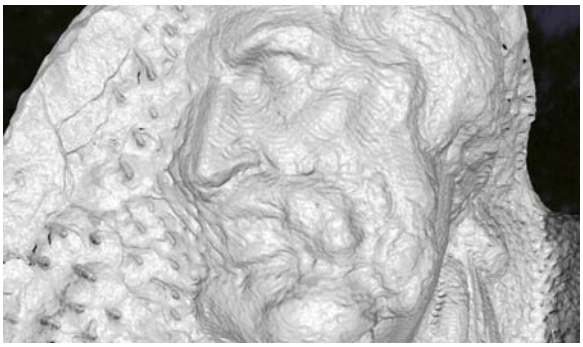

Figure 1: **Scan of Michelangelo's St. Matthew:** We precompute a cache-oblivious layout of this 9.6GB scanned model with 372M triangles. Our novel metric results in a cache-oblivious layout and at runtime reduces the vertex cache misses by more than a factor of four for interactive view-dependent rendering. As a result, we improve the frame rate by almost five times. We achieve a throughput of 106M tri/sec (at 82 fps) on an NVIDIA GeForce 6800 GPU.

Many layout algorithms and representations have been proposed for optimizing the cache access patterns for specific applications. The representations include *rendering sequences* (e.g. triangle strips) that are used to improve the rendering performance of large meshes on GPUs. Recent extensions include *processing sequences* (e.g. streaming meshes), which work well for applications that can access the data in a fixed order. Some algorithms for image processing and visualization of large datasets use space filling curves as a heuristic to improve cache coherence of a layout. These algorithms work well on models with a regular structure; however, they do not take into account the topological structure of a mesh and are not general enough to handle unstructured datasets.

Main Results: We present a novel method to compute cache-oblivious layouts of large triangle meshes. Our approach is general in terms of handling all kinds of polygonal models and cache-oblivious as it does not require any knowledge of the cache parameters or block sizes of the memory hierarchy involved in the computation.

We represent the mesh as an undirected graph $G = (V, E)$, where $|V| = n$ is the number of vertices. The mesh layout problem reduces to computing an optimal one-to-one mapping of vertices to positions in the layout, $\varphi : V \rightarrow \{1, ..., n\}$, that reduces the number of cache misses. Our specific contributions include:

1. Deriving a practical cache-oblivious metric that estimates the number of cache misses.
2. Transforming the layout computation to an optimization problem based on our metric.
3. Solving the combinatorial optimization problem using a multilevel minimization algorithm.

We also extend our graph-based formulation to compute cache-oblivious layouts of bounding volume and multiresolution hierarchies of large meshes.

We use cache-oblivious layouts for three applications: view-dependent rendering of massive models, collision detection between complex models, and isocontour extraction. In order to show the generality of our approach, we compute layouts of several kinds of geometric models including CAD environments, scanned models, isosurfaces, and terrains. We use these layouts directly without any modification to the runtime application. Our layouts significantly

reduce the number of cache misses and improve the overall performance. Compared to a variety of popular mesh layouts, we achieve on average:

1. Over an order of magnitude improvement in performance for isocontour extraction.
2. A five time improvement in rendering throughput for view-dependent rendering of multi-resolution meshes.
3. A two time speedup in collision detection queries based on bounding volume hierarchies.

Organization: The rest of the paper is organized as follows. We give a brief summary of related work on cache-efficient algorithms and mesh layouts in Section 2. Section 3 gives an overview of our approach and presents techniques for computing the graph layout of hierarchical representations. We present our cache-oblivious metric in Section 4 and describe the multilevel optimization algorithm for computing the layouts in Section 5. Section 6 highlights the use of our layouts in three different applications. We analyze our algorithms and discuss some of their limitations in Section 7.

2 Related Work

In this section we briefly review related work on cache-efficient algorithms, out-of-core techniques, mesh sequences, and layouts.

2.1 Cache-Efficient Algorithms

Cache-efficient algorithms have received considerable attention over last two decades in theoretical computer science and compiler literature. These algorithms include theoretical models of cache behavior [Vitter 2001; Sen et al. 2002], and compiler optimizations based on tiling, strip-mining, and loop interchanging; all of these can minimize cache misses [Coleman and McKinley 1995].

At a high level, cache-efficient algorithms can be classified as either cache-aware or cache-oblivious. Cache-aware algorithms utilize knowledge of cache parameters, such as cache block size [Vitter 2001]. On the other hand, cache-oblivious algorithms do not assume any knowledge of cache parameters [Frigo et al. 1999]. There is a considerable amount of literature on developing cache-efficient algorithms for specific problems and applications, including numerical programs, sorting, geometric computations, matrix multiplication, FFT, and graph algorithms. Most of these algorithms reorganize the data structures for the underlying application, i.e., computation reordering. More details are given in recent surveys [Arge et al. 2004; Vitter 2001]. There exists relatively little work on computing cache-coherent layouts for a wide variety of applications.

2.2 Out-of-Core Mesh Processing

Out-of-core algorithms are designed to handle massive geometric datasets on computers with finite memory. A recent survey of these algorithms and their applications is given in [Silva et al. 2002]. The survey includes techniques for efficient disk layouts that reduce the number of disk accesses and the time taken to load the data required at runtime. These algorithms have been used for model simplification [Cignoni et al. 2003], interactive display of large datasets composed of point primitives [Rusinkiewicz and Levoy 2000] or polygons [Corrêa et al. 2003; Yoon et al. 2004b], model compression [Isenburg and Gumhold 2003], and collision detection [Franquesa-Niubo and Brunet 2003; Wilson et al. 1999]. Out-of-core techniques are complimentary to cache-coherent mesh layouts.

2.3 Mesh Sequences and Layouts

The order in which a mesh is laid out can affect the performance of algorithms operating on the mesh. Several possibilities have been considered.

Rendering Sequences: Modern GPUs maintain a small buffer to reuse recently accessed vertices. In order to maximize the benefits of vertex buffers for fast rendering, triangle reordering is necessary. This approach was pioneered by Deering [1995]. The resulting ordering of triangles is called a triangle strip or a rendering sequence.

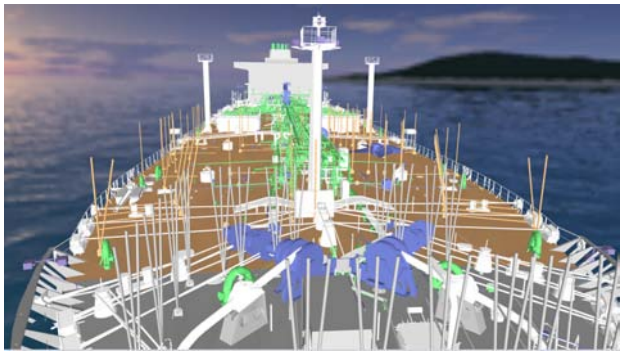

Figure 2: **Double Eagle Tanker:** We compute a cache-oblivious layout of the tanker with 82M triangles and more than 127K different objects. This model has an irregular distribution of primitives. We use our layout to reduce vertex cache misses and to improve the frame rate for interactive view-dependent rendering by a factor of two; we achieve a throughput of 47M tri/sec (at 35 fps) on an NVIDIA GeForce 6800 GPU.

Hoppe [1999] casts the triangle reordering as a discrete optimization problem with a cost function relying on a specific vertex buffer size. If a triangle mesh is computed on the fly using view-dependent simplification or other geometric operations, the rendering sequences need to be recomputed to maintain high throughput. Other techniques improve the rendering performance of view-dependent algorithms by computing rendering sequences not tailored to a particular cache size [Bogomjakov and Gotsman 2002; Karni et al. 2002]. However, these algorithms have been applied only to relatively small models (e.g., 100K triangles).

Processing Sequences: Isenburg and Gumhold [2003] propose processing sequences as an extension of rendering sequences to large-data processing. A processing sequence represents a mesh as an interleaved ordering of indexed triangles and vertices that can be streamed through main memory [Isenburg and Lindstrom 2004]. However, global mesh access is restricted to a fixed traversal order; only localized random access to the buffered part of the mesh is supported as it streams through memory. This representation is mostly useful for offline applications (e.g., simplification and compression) that can adapt their computations to the fixed ordering.

Space Filling Curves: Many algorithms use space filling curves [Sagan 1994] to compute cache-friendly layouts of volumetric grids or height fields. These layouts are widely used to improve performance of image processing [Velho and Gomes 1991] and terrain or volume visualization [Pascucci and Frank 2001; Lindstrom and Pascucci 2001]. A standard method of constructing a layout is to embed the meshes or geometric objects in a uniform structure that contains the space filling curve. Therefore, these algorithms have been used for objects or meshes with a regular structure (e.g. images and height fields). Methods based on space filling curves do not consider the topological structure of meshes. It is unclear whether these approaches would extend to large CAD environments with an irregular distribution of geometric primitives. Moreover, if an application needs to access the mesh primitives based on connectivity information, space filling curves may not be useful. Algorithms have also been proposed to compute paths on constrained, unstructured graphs as well as to generate triangle strips and finite-element mesh layouts [Heber et al. 2000; Oliker et al. 2002; Bartholdi and Goldsman 2004; Gopi and Eppstein 2004].

Sparse Matrix Reordering: There is considerable research on converting sparse matrices into banded ones to improve the performance of various matrix operations [Diaz et al. 2002]. Common graph and matrix reordering algorithms attempt to minimize one of three measures: bandwidth (maximum edge length), profile (sum of maximum per-vertex edge length), and wavefront (maximum front size, as in stream processing). These measures are closely connected with MLA and layouts for streaming, and generally are more applicable to stream layout than cache-oblivious mesh layout.

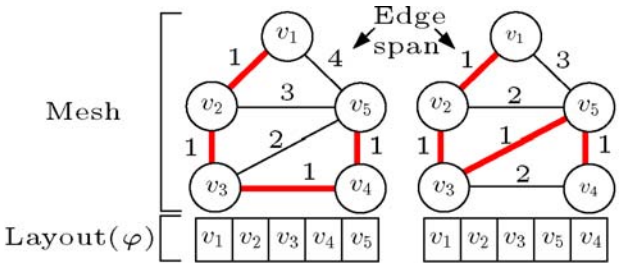

Figure 3: **Vertex layout for a mesh:** A mesh consisting of 5 vertices is shown with two different orderings obtained using a local permutation of v_4 and v_5. We highlight the span of each edge based on the layout. The ordering shown on the right minimizes cache misses according to our cache-oblivious metric.

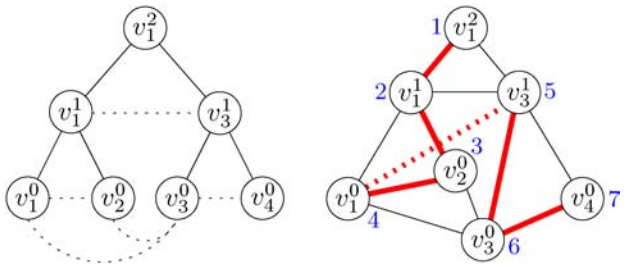

Figure 4: **A layout of a vertex hierarchy:** A vertex hierarchy is shown on the left. Each node of the vertex hierarchy represents a leaf or intermediate level vertex. A parent node, v_1^1, is constructed by merging two child nodes, v_1^0 and v_2^0. Solid lines between the nodes represent connectivity access and dotted lines represent the spatial locality between the nodes at the same level. Its corresponding graph and a layout of the vertices (with a position in the layout shown in blue) are shown on the right.

3 Mesh Layout and Cache Misses

In this section, we introduce some of the terminology used in the rest of the paper and give an overview of memory hierarchies. We represent a mesh as a graph and extend our approach to layouts of multi-resolution and bounding volume hierarchies of a mesh.

3.1 Memory Hierarchy and Caches

Most modern computers use hierarchies of memory levels, where each level of memory serves as a *cache* for the next level. Memory hierarchies have two main characteristics. First, higher levels are larger in size and farther from the processor, and they have slower access times. Second, data is moved in large blocks between different memory levels. The mesh layout is initially stored in the highest memory level, typically the disk. The portion of the layout accessed by the application is transferred in large blocks into the next lower level, such as main memory. A transfer is performed whenever there is a cache miss between two adjacent levels of the memory hierarchy. The number of cache misses is dependent on the layout of the original mesh in memory and the access pattern of the application.

3.2 Mesh Layout

A mesh layout is a linear sequence of vertices and triangles of the mesh. We construct a graph in which each vertex represents a data element of the mesh. An edge exists between two vertices of the graph if their representative data elements are likely to be accessed in succession by an application at runtime.

For single-resolution mesh layout, we map mesh vertices and edges to graph vertices and edges. A vertex layout of an undirected graph $G = (V, E)$ is a one-to-one mapping of vertices to positions, $\varphi : V \rightarrow \{1, \ldots, n\}$, where $|V| = n$. Our goal is to find a mapping, φ, that minimizes the number of cache misses during accesses to the mesh.

A mesh layout is composed of two layouts: a vertex layout and a triangle layout. While a triangle layout can be constructed as a vertex layout of the dual graph, because the triangles and their vertices are often accessed together we ensure that the triangle layout is "compatible" with the vertex layout, e.g. by ordering triangles on their minimum or maximum vertex index (cf. [Isenburg and Lindstrom 2004]). In the rest of the paper, we use the term layout to refer to a vertex layout for the sake of clarity.

3.3 Layouts of Multiresolution Meshes and Hierarchies

In this section, we show that our graph-based formulation can be used to compute cache-coherent layouts of hierarchical representations. Hierarchical data structures are widely used to speed up computations on large meshes. Two types of hierarchies are used for geometric processing and interactive visualization: bounding volume hierarchies (BVHs) and multi-resolution hierarchies (MRHs). The BVHs use simple bounding shapes (e.g. spheres, AABBs, OBBs) to enclose a group of triangles in a hierarchical manner. MRHs are used to generate a simplification or approximation of the original model based on an error metric; these include vertex hierarchies (VHs) used for view-dependent rendering, and hierarchies that are defined using subdivision rules.

Terminology: We define $v_i = v_i^0$ as the ith vertex at the leaf level of the hierarchy, and v_i^k as a vertex at the kth level. v_i^k is a parent of v_i^{k-1} and v_{i+1}^{k-1}. In the case of a BVH, v_i^k denotes a bounding volume. In the case of a vertex hierarchy, v_i^k denotes a vertex generated by decimation operations. An example of a vertex hierarchy is shown in Fig. 4.

In order to compute a layout of a hierarchy, we construct a graph that captures cache-coherent access patterns to the hierarchy. We add extra edges to our graph that capture the spatial locality and parent-child relationships within the hierarchy.

1. **Connectivity between parent-children nodes:** Once a node of a hierarchy is accessed, it is highly likely that its parent or child nodes would be accessed soon. For example, a vertex-split of a node in the VH activates its child nodes and an edge-collapse of two sibling nodes activates their parent node.
2. **Spatial locality between vertices at the same level:** Whenever a node is accessed, other nodes in close proximity are also highly likely to be accessed thereafter. For example, collisions or contacts between two objects occur in small localized regions of a mesh. Therefore, if a node of a BVH is activated, other nearby nodes are either colliding or are in close proximity and may be accessed soon.

Graph Representation: We take these localities into account and compute an undirected graph for MRHs and BVHs. For a BVH, we represent each BV with a separate vertex in the graph. The edges in our graph include edges between parent vertices and their children, and edges between nearby vertices at each level of the BVH. Edges are created between nearby vertices when their Euclidean distance falls below a given threshold. Fig. 4 shows the graph as well as its layout for the given vertex hierarchy. More details on connectivity and spatial localities of BVHs are also available [Yoon and Manocha 2005].

4 Cache-Oblivious Layouts

In this section we present a novel algorithm for computing a cache-coherent layout of a mesh. We make no assumptions about cache parameters and compute the layout in a cache-oblivious manner.

4.1 Terminology

We use the following terminology in the rest of the paper. The *edge span* of the edge between v_i and v_j in a layout is the absolute difference of the vertex indices, $|i - j|$ (see Fig. 3). We use E_l to denote the set that consists of all the edges of edge span l, where $l \in [1, n-1]$. The *edge span distribution* of a layout is the histogram of spans of all the edges in the layout. The *cache miss ratio* is the ratio of the number of cache misses to the number of accesses. The *cache miss ratio function (CMRF)*, p_l, is a function that relates the cache miss ratio to an edge span, l. The CMRF always lies within the interval $[0, 1]$; it is exactly 0 when there are no cache misses, and equals 1 when every access results in a cache miss. We alter the

Figure 5: **Puget Sound contour line:** This image shows a contour line (in black) extracted from an unstructured terrain model of the Puget Sound. The terrain is simplified down to 143M triangles. We extracted the largest component (223K edges) of the level set at 500 meters of elevation. Our cache-oblivious layouts improve the performance of the isocontour extraction algorithm by more than an order of magnitude.

layouts using a *local permutation* that reorders a small subset of the vertices. The local permutation changes the edge span of edges that are incident to the affected vertices (see Fig. 3).

4.2 Metrics for Cache Misses

We first define a metric for estimating the cache misses for a given layout. One well known metric for the graph layout problem is the minimum linear arrangement (MLA), which minimizes the sum of edge spans [Diaz et al. 2002]. Heuristics for the NP-hard MLA problem, such as spectral sequencing, have been used to compute mesh layouts for rendering and processing sequences [Bogomjakov and Gotsman 2002; Isenburg and Lindstrom 2004]. We have empirically observed that metrics used to estimate MLA may not minimize cache misses for general applications (See Fig. 6). This is mostly because MLA results in a front-advancing sweep over the mesh along a dominant direction that tends to minimize the length of the front. On a rectilinear grid, for example, MLA roughly corresponds to a row-by-row layout, which has poor worst-case performance when accessing the grid column by column. We present an alternate metric based on the edge span distribution and the CMRF that captures the locality for various access patterns and results in layouts with a more "space filling" quality. Contrary to MLA, our layouts are not biased towards a particular traversal direction.

Cache-coherent Access Pattern: If we know the runtime access pattern of a given application a priori and the CMRFs, we can compute the exact number of cache misses. However, we make no assumptions about the application and instead use a probabilistic model to estimate the number of cache misses. Our model approximates the edge span distribution of the runtime access pattern of the vertices with the edge span distribution of the layout. Based on this model, we define the expected number of cache misses of the layout as:

$$ECM = \sum_{i=1}^{n-1} |E_i| p_i \tag{1}$$

where $|E_i|$ is the cardinality of E_i and is a function of the layout, φ.

4.3 Assumptions

Our goal is to compute a layout, φ, that minimizes the expected number of cache misses for all possible cache parameters. We present a metric that is used to check whether a local permutation would reduce cache misses. We make two assumptions with respect to CMRFs: invariance and monotonicity.

Invariance: We assume that the CMRF of a layout is invariant before and after a local permutation. Since a local permutation affects only a small region of a mesh, the changes in CMRF due to a local permutation are very small.

Monotonicity: We assume that the CMRF is a monotonically non-decreasing function of edge span. As we access vertices that are farther away from the current vertex (i.e. the edge spans increase), the probability of having a cache miss increases, until eventually leveling off at 1.

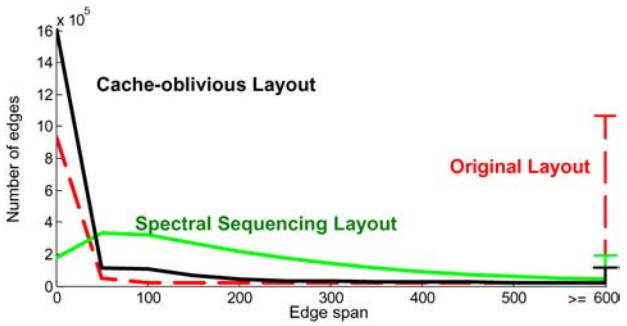

Figure 6: **Edge span distributions:** The edge span histogram of the dragon model with 871K triangles and 437K vertices. We show the histogram of the original model representation (red), spectral sequencing (green), and our cache-oblivious metric (black). In the original layout, a large number of edges have edge spans greater than 600. Intuitively, our cache-oblivious metric favors edges that have small edge spans. Therefore, our layouts reduce cache misses.

4.4 Cache-oblivious Metric

Our cache-oblivious metric is used to decide whether a local permutation decreases the expected number of cache misses, which due to the invariance of p_i is true if the following inequality holds:

$$\sum_{i=1}^{n-1} (|E_i| + \Delta|E_i|) p_i < \sum_{i=1}^{n-1} |E_i| p_i \Leftrightarrow \sum_{j=1}^{m} \Delta|E_{l(j)}| p_{l(j)} < 0 \tag{2}$$

Here $\Delta|E_i|$ is the signed change in the number of edges with edge span i after a local permutation and $n-1$ is maximum edge span for a mesh with n vertices. Furthermore, we let m denote the number of sets (among $E_1, E_2, \ldots, E_{n-1}$) whose cardinality changes because of the permutation, and let $l(j)$ denote the edge span associated with the jth such set, with $l(j) < l(j+1)$ and $m \ll n-1$.

Constant Edge Property: The total number of edges in a layout is the same before and after the local permutation. Hence

$$\sum_{j=1}^{m} \Delta|E_{l(j)}| = 0 \tag{3}$$

Parameterization of cache miss ratio: We parameterize each cache miss ratio, $p_{l(j)}$, by introducing a parametric variable, x_j, which due to the monotonicity of $p_{l(j)}$ is monotonically non-decreasing with j. This is represented as:

$$p_{l(j)} = x_j p_{l(1)} \tag{4}$$

where $1 \le j \le m$ and

$$1 = x_1 \le x_2 \le \cdots \le x_{m-1} \le x_m \le \frac{1}{p_{l(1)}} \tag{5}$$

$p_{l(1)}$ is the cache miss ratio of the first edge, and $0 \le p_{l(1)} \le 1$.

The leftmost constraint of Eq. (5) is obvious because $p_{l(1)} = x_1 p_{l(1)}$. The rightmost constraint is computed from $p_{l(m)} = x_m p_{l(1)} \le 1$, because all the cache miss values are less than or equal to 1.

By substituting the parameterization of cache miss ratios shown in Eq. (4) into Eq. (2) and canceling the constant $p_{l(1)}$, we have:

$$\sum_{j=1}^{m} \Delta|E_{l(j)}| x_j < 0. \tag{6}$$

This is our exact cache-oblivious metric.

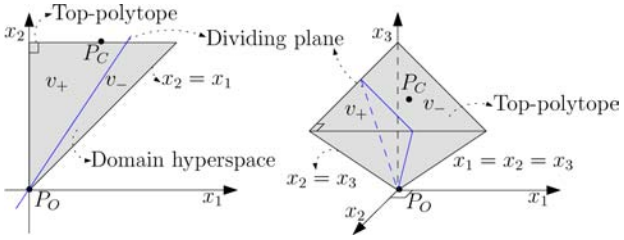

Figure 7: **Geometric volume computation:** The left figure shows a 2D geometric view of Eq. (6). The 3D version is shown in the right figure.

4.5 Geometric Formulation

We reduce the computation of the expression in Eq. (6) to a geometric volume computation in an m dimensional hyperspace. Geometrically, the parameterization domain represented in Eq. (5) defines a closed hyperspace in $\mathbb{R}^m$. We refer to this hyperspace as the *domain*. Eq. (6) defines a closed subspace within the domain of Eq. (5). Moreover, a dividing hyperplane defining this closed subspace passes through the point, $\{1, 1, \ldots, 1\} = P_O \in \mathbb{R}^m$, of the domain according to the constant edge property highlighted in Eq. (3). We also define the *top-polytope* of the domain as the polytope intersecting the rightmost constraints of Eq. (5) with the open hyperspace defined by the other constraints of Eq. (5). Moreover, we define V_+ to be the volume of the subspace represented in Eq. (6) and V_- to be the volume of its complement within the closed domain. These geometric concepts in 2 and 3 dimensions are illustrated in Fig. 7.

Volume Computation: Intuitively speaking, the volume V_+ corresponds to the set of cache configurations parameterized by $\{x_j\}$ for which we expect a reduction in cache misses. Since we assume all configurations to be equally likely, we probabilistically reduce the number of cache misses by accepting a local permutation whenever V_+ is larger than V_-.

Complexity of Volume Computation: The computation of the volume of a convex polytope defined by $m + 1$ hyperplanes in m dimensions is a hard problem. The complexity of exact volume computation is $O(m^{m+1})$ [Lasserre and Zeron 2001] and an approximate algorithm of complexity $O(m^5)$ is presented in [Kannan et al. 1997]. In our application, each local permutation involves around 20–50 edges and these algorithms can be rather slow.

4.6 Fast and Approximate Metric

Given the complexity of exact volume computation, we use an approximate metric to check whether a local permutation would reduce the expected number of cache misses. In particular, we use a single sample point—the centroid of the top-polytope—as an estimate of $\{x_j\}$ and compute an approximate metric with low error.

Note that the dividing hyperplane between V_+ and V_- passes through the point P_O. Therefore, the ratio of V_+ to V_- is equal to the ratio of the $(m-1)$ dimensional areas formed by partitioning the top-polytope by the same dividing hyperplane. For example, in the 2D case, the result of volume comparison computed by substituting a centroid into Eq. (6) is exactly same as the result of the 2D area comparison between V_+ and V_-. This formulation extends to 3D, but it introduces some error. The error is maximized when the dividing plane is parallel to one of the edges of the top-polytope and it is minimized (i.e., exactly zero) when the plane passes through one of its vertices.

We generalize this idea to m dimensions. P_C, the centroid of a top-polytope, is defined as $(\frac{1}{m}, \frac{2}{m}, \ldots, \frac{m-1}{m}, \frac{m}{m}) \times \frac{1}{p_{l(1)}} + P_O$. By substituting P_C into Eq. 6 and canceling the constants, $\frac{1}{p_{l(1)}}$ and P_O, we have:

$$\sum_{j=1}^{m} \Delta |E_{l(j)}| j < 0 \qquad (7)$$

If inequality (7) holds, we allow the local permutation. Based on this metric, we compute a layout, φ, that minimizes the number of cache misses.

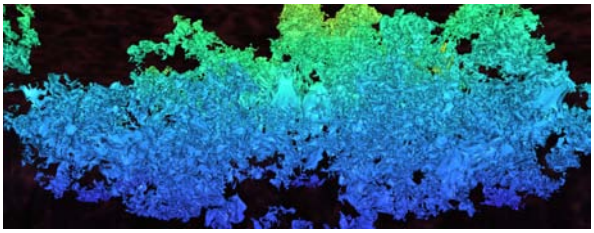

Figure 8: **Isosurface model:** This image shows a complex isosurface (100M triangles) generated from a 3D simulation of turbulent fluids mixing. Our layout reduces the vertex cache misses by more than a factor of four during view-dependent rendering. As a result, we improve the frame rate by 4 times as compared to prior approaches. We achieve a throughput of 90M tri/sec (at 30 fps) on a PC with an NVIDIA GeForce 6800 GPU.

Error Bounds on Approximate Metric: The approximate cache-oblivious metric has a worst case error of 26%, when the dividing hyperplane is parallel to one of the edges of the top-polytope. In practice, the worst case configuration is rare. In our benchmarks, we found that the actual error is typically much less than the worst case bound.

5 Layout Optimization

Given the cache-oblivious metric, our goal is to find the layout, φ, that minimizes the expected number of cache misses, defined in Eq. (1). This is a combinatorial optimization problem for graph layouts [Diaz et al. 2002]. Finding a globally optimal layout is NP-hard [Garey et al. 1976] due to the large number of permutations of the set of vertices. Instead, we use a heuristic based on *multilevel minimization* that performs local permutations to compute a locally optimal layout.

5.1 Multilevel Minimization

Our multilevel algorithm consists of three main steps. First, a series of coarsening operations on the graph are computed. Next, we compute an ordering of vertices of the coarsest graph. Finally, we recursively expand the graph by reversing the coarsening operations and refine the ordering by performing *local permutations*. We will now describe each of these steps in more detail.

Coarsening Step: The goal of the coarsening phase is to cluster vertices in order to reduce the size of the graph while preserving the essential properties needed to compute a good layout. We have tried two approaches: clustering via graph partitioning [Karypis and Kumar 1998] and via streaming edge-collapse [Isenburg and Lindstrom 2004], using only the topological structure of the graph as criterion for collapsing edges. As mentioned above, geometric locality can be preserved by adding additional edges to the graph between spatially close vertices.

Ordering Step: Given the coarsest graph of a handful of vertices, we list all possible orderings of its vertices and compute the costs based on the cache-oblivious metric from Eq. (7). We choose a vertex ordering that has the minimum cost among all possible orderings.

Refinement Step: We reverse the sequence of coarsening operations applied earlier and exhaustively compute the locally optimal permutation of the subset of vertices involved in each corresponding refinement operation.

5.2 Local Permutation

We compute local permutations of the vertices during the ordering and refinement steps. A local permutation affects only a small number of vertices in the layout and changes the edge spans of those edges that are incident to these vertices. Therefore, we can efficiently recompute the cost associated with the metric. Each local permutation involves $k!$ possible orderings for k vertices, which during refinement replace their common parent in the evolving layout. For efficiency we restrict each coarsening operation to merge no more than $k = 5$ vertices at a time, and also limit the number of vertices in the coarsest graph to 5.

Model	Type	Vert. (M)	Tri. (M)	Size (MB)	Layout Comp. (min)
Dragon	s	0.4	0.8	33	0.25
Lucy	s	14.0	28.0	520	8
David	s	28.0	56.0	700	19
Double Eagle	c	77.7	81.7	3,346	56
Isosurface	i	50.5	100.0	2,543	49
Puget Sound	t	67.0	134.0	1,675	58
St. Matthew	s	186.0	372.0	9,611	176
Atlas	s	254.0	507.0	12,422	244

Table 1: **Layout Benchmarks:** Model complexity and time spent on layout computation are shown. Type indicates model type: s for scanned model, i for isosurface, c for CAD model, and t for terrain model. Vert. is the number of vertices and Tri. is the number of triangles of a model. Layout Comp. is time spent on layout computation.

Model	Double Eagle	Isosurface	St. Matthew
PoE	3	5	1
Frame rate	35	30	82
Rendering throughput(million tri./sec.)	47	90	106
Avg. Improvement	2.1	4.5	4.6
ACMR	1.58	0.75	0.72

Table 2: **View-Dependent Rendering** This table highlights the frame rate and rendering throughput for different models. We improve the rendering throughput and frame rates by $2.1 - 4.6$ times. The ACMR was computed with a buffer consisting of 24 vertices.

5.3 Out-of-Core Multilevel Optimization

The multilevel optimization algorithm needs to maintain an ordering of vertices along with a series of coarsening operations. For large meshes composed of hundreds of millions of vertices, it may not be possible to store all this information in main memory. In both of our graph partitioning and edge-collapse approaches, we compute a set of clusters, each containing a subset of vertices. Each cluster represents a subgraph and we compute an inter-cluster ordering among the clusters. We then follow the cluster ordering and compute a layout of all the vertices within each cluster using our multilevel minimization algorithm.

6 Implementation and Performance

In this section we describe our implementation and use cache coherent layouts to improve the performance of three applications: view-dependent rendering of massive models, collision detection between complex models, and isocontour extraction. Moreover, we used different kinds of models including CAD environments, scanned datasets, terrains, and isosurfaces to test the performance of cache coherent layouts. We also compare the performance of our metric with other metrics used for mesh layout.

6.1 Implementation

We have implemented our layout computation and out-of-core view-dependent rendering and collision detection algorithms on a 2.4GHz Pentium-4 PC with 1GB of RAM and a GeForce Ultra FX 6800 GPU with 256MB of video memory.

We use the METIS graph partitioning library [Karypis and Kumar 1998] for coarsening operations to lay out vertex and bounding volume hierarchies. Our current unoptimized implementation of the out-of-core layout computation processes about 30K triangles per sec. In the case of the St. Matthew model, our second largest dataset, layout computation takes about 2.6 hours.

Memory-mapped I/O: Our system runs on Windows XP and uses the operating system's virtual memory through memory mapped files [Lindstrom and Pascucci 2001]. Windows XP can map only up to 2GB of user-addressable space. We overcome this limitation by mapping a small portion of the file at a time and remapping when data is required from outside this range.

6.2 View-dependent rendering

View-dependent rendering and simplification are frequently used for interactive display of massive models. These algorithms precompute a multiresolution hierarchy of a large model (e.g. a vertex hierarchy). At runtime, a dynamic simplification of the model is computed by incrementally traversing the hierarchy until the desired pixels of error (PoE) tolerance in image space is met. Current view-dependent rendering algorithms are unable to achieve high polygon rendering throughput on the GPU for massive models composed of tens or hundreds of millions of triangles. It is not possible to compute rendering sequences at interactive rates for such massive models.

PoE	0.75	1	4	20
COL	0.71	0.72	0.73	0.74
SL	2.85	2.85	2.92	2.96

Table 3: **ACMR vs. PoE:** ACMRs are computed as we increase the PoE, i.e. use a more drastic simplification. The ACMRs of cache-oblivious layouts (COL) are still low even when a higher PoE is selected.

We use a clustered hierarchy of progressive meshes (CHPM) representation [Yoon et al. 2004b] for view-dependent refinement along with occlusion culling and out-of-core data management. The CHPM-based refinement algorithm is very fast and most of the frame time is spent in rendering the simplified model. We precompute a cache-oblivious layout (COL) of the CHPM and use it to reduce the cache misses for the vertex cache on the GPU. We computed layouts for three massive models including a CAD environment of a tanker with 127K separate objects (Fig. 2), a scanned model of St. Matthew (Fig. 1) and an isosurface model (Fig. 8). The details of these models are summarized in Table 1. We measured the performance of our algorithm along paths through the models. These paths are shown in the accompanying video.

6.2.1 Results

Table 2 highlights the benefit of COL over the simplification layout (SL), whose vertex layout and triangle layout are computed by the underlying simplification algorithm. We are able to increase the rendering throughput by a factor of of 2.1-4.6 times by precomputing a COL of the CHPM of each model. We obtain a rendering throughout of 106M triangles per second on average, with a peak performance of 145M triangles per second.

Average Cache Miss Ratio (ACMR): The ACMR is defined by the ratio of the number of accessed vertices to the number of rendered triangles for a particular vertex cache size [Hoppe 1999]. If the number of triangles in the model is roughly twice the number of vertices (e.g. the St. Matthew and isosurface models), then the ACMR is within the interval $[0.5, 3]$. Therefore, the theoretical upper bound on cache miss reduction is a factor of 6. For a cache of 24 vertices, we improve the ACMR by a factor of 3.95 and get a 4.5 times speedup in the rendering throughput. On the other hand, if the number of vertices in the model is roughly the same as the number of triangles, as in the tanker model, then the ACMR is within the interval $[1, 3]$ and the upper bound on cache miss reduction is 3 times. For this model, we improve the ACMR by a factor of 1.89 and the rendering throughput by a factor of 2.1. To verify the cache-oblivious nature of our layouts, we also simulated a FIFO vertex cache of configurable size and measured the ACMR as a function of cache size (Fig. 10). Table 3 shows the ACMR achieved by varying the PoE in the St. Matthew model.

6.3 Collision Detection

Many collision detection algorithms use bounding volume hierarchies to accelerate the interference computations [Lin and Manocha 2003]. We use cache-oblivious layouts to improve the performance of collision detection algorithms. In particular, we compute layouts of OBB-trees [Gottschalk et al. 1996] and use them to accelerate collision queries within a dynamic simulator. The collision detection algorithm traverses the bounding volume hierarchy of each model and checks for overlap between the OBBs and triangle pairs. We have tested the performance of our collision detection algorithm in a rigid body simulation where 20 dragons (800K triangles each) drop on the Lucy model (28M triangles). The details of these models are shown in Table 1. Fig. 9 shows a snapshot from our simulation.

We compared the performance of our cache-oblivious layout with the RAPID library [Gottschalk et al. 1996]. The OBBs are precomputed and stored in memory-mapped files and only the ordering of the hierarchy is modified. We compared our cache-oblivious layout with a depth-first layout (DFL) of OBB-trees. The DFL is computed

Figure 9: **Dynamic Simulation:** Dragons consisting of 800K triangles are dropping on the Lucy model consisting of 28M triangles. We obtain 2 times improvement by using COL on average.

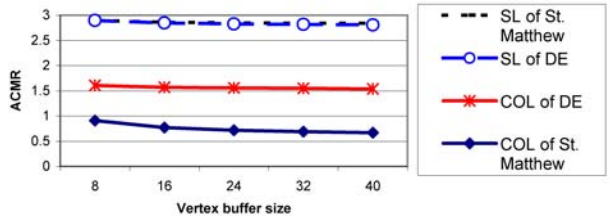

Figure 10: **ACMR vs. cache size:** ACMRs of cache-oblivious layout (COL) and simplification layout (SL) of the St. Matthew and double eagle tanker are shown. As the cache size increases, the improvement of COL becomes larger, and is 3.95 at a cache size of 24 in the St. Matthew model. Note that the lower bound on ACMR is 0.5 in St. Matthew and 1 in the double eagle tanker. The two SL curves almost overlap.

by traversing the hierarchy from its root node in a depth-first order. We chose DFL because it preserves the spatial locality within the bounding volume hierarchy.

6.3.1 Results

We are able to achieve 2 times improvement in performance over the depth-first layout on average. This is mainly due to reduced cache misses, including main memory page faults. We observe more than 2 times improvement whenever there are more broad contact regions. Such contacts trigger a higher number of page faults and in such situations we obtain a higher benefit from cache-oblivious layouts. The query times of collision detection during the dynamic simulation are shown in Fig. 11.

6.4 Isocontour Extraction

The problem of extracting an isocontour from an unstructured dataset frequently arises in geographic information systems and scientific visualization. We use an algorithm based on seeds sets [van Kreveld et al. 1997] to extract the isocontour of a single-resolution mesh. The running time of this algorithm is dominated by the traversal of the triangles intersecting the contour itself.

We use this algorithm to extract isocontours of a large terrain (Fig. 5) and compute equivalent geometric queries such as extracting ridge lines of a terrain[1] and cross-sections of large geometric models.

6.4.1 Comparison with other layouts

We compare the the performance of the isocontouring algorithm on four models, each stored in five different layouts. In addition to our cache-oblivious layout, we also store the meshes in geometric $X/Y/Z$ orders (vertices sorted by their position along the corresponding coordinate axis) and in spectral sequencing order [Diaz et al. 2002]. We use edge-collapse as the coarsening step for computing cache-oblivious layouts and store all meshes in a streaming format [Isenburg and Lindstrom 2004], which allows us to quickly compute the on-disk mesh data structure in a preprocess.

Table 4 reports the time in seconds to compute an isocontour and a ridge line for the terrain models and to compute cross-sections of the other 3D models. The tests have been performed on a 1.3GHz Itanium Linux PC with 2GB of main memory. We take advantage of the 64-bit architecture and memory map the entire model. We do not perform any explicit paging. This way we ensure that we do not bias the results in favor of any particular layout.

[1]For extracting a ridge line the seed point is a saddle and the propagation goes upward to the closest maxima instead of following an isocontour.

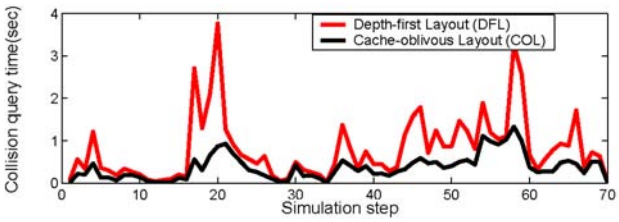

Figure 11: **Performance of Collision Detection:** Average query times for collision detection between the Lucy model and the dragon model with COL and DFL are shown. We obtain 2 times improvement in the query time on average.

Model	Puget Sound		Lucy	David	Atlas
Out. edg.	223K (Contour)	14K (Ridge)	17K (Section)	22K (Section)	38K (Section)
Cac. Obl.	026 (000.5)	003 (000.03)	03.3 (.04)	05.9 (.057)	010 (000.09)
Geom. X	232 (227.8)	001 (000.04)	01.2 (.04)	00.2 (.051)	015 (000.09)
Geom. Y	218 (215.5)	195 (185.10)	39.1 (.09)	60.7 (.103)	419 (379.78)
Geom. Z	011 (000.6)	135 (113.81)	26.1 (.09)	45.5 (.102)	443 (382.60)
Spec. Seq.	150 (127.3)	023 (000.04)	21.0 (.06)	43.1 (.068)	088 (000.10)

Table 4: **Isocontouring.** Time in seconds (on a 1.3GHz linux PC with 2GB of memory) for extracting an isocontour (or equivalent geometric queries) for several models stored each in five different mesh layouts: cache-oblivious, with vertices sorted by X/Y/Z geometric coordinate, and spectral sequencing. In parentheses we report the time for second immediate re-computation of the same contour when all the cache levels in the memory hierarchy have been loaded. In all cases, the performance of our cache-oblivious layout is comparable to the one optimized for the particular geometric query. This demonstrates the benefit of our layout for general applications.

The empirical data shows that our cache-oblivious layout minimizes the worst case cost of generic coherent traversals. The three layouts that are sorted by geometric direction along the X, Y, and Z axis show that the worst case performance is at least one order of magnitude slower than the best case, which is achieved by the layout that happens to be perfectly aligned along the query direction. The spectral sequencing layout also does not perform well since the geometric query is unlikely to follow its streaming order. Our cache-oblivious layout consistently exhibits good performance.

The running times reported in parentheses in Table 4 are for a second immediate re-computation of the same contour, ridge line, or cross-section. They demonstrate the performance when all the cache levels have been loaded by the first computation. In this case our cache-oblivious layout is always as fast as the the best case and can be two times to several orders of magnitude faster than the worst case. More importantly, this test demonstrates the cache-oblivious nature of the approach since performance advantages at different scales are achieved both when disk paging is necessary and when only internal memory and L2 caches are involved. In case of the Puget Sound terrain model, our cache-oblivious layout is the only layout that takes advantage of loaded cache levels for both the queries (i.e., isocontour and ridge line extraction).

7 Analysis and Limitations

We present a novel metric based on edge span distribution and CMRF to determine whether a local permutation on a layout reduces the expected number of cache misses. In practice, our algorithm computes layouts for which a high fraction of edges have very small edge spans. At the same time, a small number of edges in the layout can have a very large edge span, as shown in Fig. 6. This distribution of edge spans improves the performance because edges with small edge span increase the probability of a cache hit, while the actual length of very high-span edges has little impact on the likelihood of a cache hit.

Our multilevel minimization algorithm is efficient and produces reasonably good results for our applications. Moreover, our minimization algorithm maps very well to out-of-core computations and is able to handle very large graphs and meshes with hundreds of millions of triangles. We have applied our cache-oblivious layouts to

models with irregular distribution of geometric primitives or irregular structure, for which prior algorithms based on space-filling curves may not work well.

Limitations: Our metric and layout computation algorithm has several limitations. The assumptions we make about invariance and monotonicity of CMRFs may not hold for all applications, and our minimization algorithm does not necessarily compute a globally optimal solution. Our cache-oblivious layouts result in good improvements primarily in applications where the running time is dominated by data access.

8 Conclusion and Future Work

We have presented a novel approach for computing cache-oblivious layouts of large meshes. We make no assumptions about the runtime access pattern and we compute an ordering that results in high locality. We show that our formulation can be extended to compute layouts of bounding volume and multiresolution hierarchies of large meshes. We use a probabilistic model to minimize the number of cache misses. Our preliminary results indicate that our metric works well in practice for reducing cache misses. Furthermore, we computed cache-oblivious layouts of different kinds of geometric datasets including scanned models, isosurfaces, terrain, and CAD environments with irregular distributions of primitives. We used our layouts to improve the performance of view-dependent rendering, collision detection and isocontour extraction by $2 - 20$ times without any modification of the algorithm or runtime applications.

There are many avenues for future work. In this paper, we have only considered cache-oblivious access and we can obtain further improvement in performance by taking into account cache parameters to design an improved metric. We would like to use our layout to improve the performance of algorithms for processing and manipulation of large meshes including simplification, compression, parameterization, smoothing, isosurface extraction, shadow generation, and approximate collision detection [Yoon et al. 2004a]. We would also like to use our graph-based formulation to compute cache-coherent layouts for other kinds of datasets, including point primitives and unstructured volumetric grids. Finally, we would like to combine data layouts with application specific techniques to increase program locality to further improve the cache access pattern and efficiency.

Acknowledgments

This work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134 and 0118743, DARPA/RDECOM Contract N61339-04-C-0043 and Intel. Some of the work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48. The St. Matthew, Lucy, and Atlas models are courtesy of the Digital Michelangelo Project at Stanford University. The isosurface model is courtesy of the LLNL ASCI VIEWS Visualization project and the Double Eagle tanker is courtesy of Newport News Shipbuilding. We would like to thank Brandon Lloyd, Brian Salomon, Avneesh Sud, Martin Isenburg, Dawoon Jung, Elise London, and the members of UNC Walkthrough and Gamma group for their feedback on an earlier draft of the paper and technical discussions.

References

ARGE, L., BRODAL, G., AND FAGERBERG, R. 2004. Cache oblivious data structures. *Handbook on Data Structures and Applications*.

BARTHOLDI, J., AND GOLDSMAN, P. 2004. Multiresolution indexing of triangulated irregular networks. In *IEEE Transaction on Visualization and Computer Graphics*, 484–495.

BOGOMJAKOV, A., AND GOTSMAN, C. 2002. Universal rendering sequences for transparent vertex caching of progressive meshes. In *Computer Graphics Forum*, 137–148.

CIGNONI, P., MONTANI, C., ROCCHINI, C., AND SCOPIGNO, R. 2003. External memory management and simplification of huge meshes. In *IEEE Transaction on Visualization and Computer Graphics*, 525–537.

COLEMAN, S., AND MCKINLEY, K. 1995. Tile size selection using cache organization and data layout. *SIGPLAN Conference on Programming Language Design and Implementation*, 279–290.

CORRÊA, W. T., KLOSOWSKI, J. T., AND SILVA, C. T. 2003. Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of PVG 2003 (6th IEEE Symposium on Parallel and Large-Data Visualization and Graphics)*, 1–8.

DEERING, M. F. 1995. Geometry compression. In *SIGGRAPH 95 Conference Proceedings*, Addison Wesley, R. Cook, Ed., Annual Conference Series, ACM SIGGRAPH, 13–20. held in Los Angeles, California, 06-11 August 1995.

DIAZ, J., PETIT, J., AND SERNA, M. 2002. A survey on graph layout problems. *ACM Computing Surveys 34*, 313–356.

FRANQUESA-NIUBO, M., AND BRUNET, P. 2003. Collision prediction using mktrees. *Proc. CEIG*, 217–232.

FRIGO, M., LEISERSON, C., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. *Symposium on Foundations of Computer Science*, 285–297.

GAREY, M. R., JOHNSON, D., AND STOCKMEYER, L. 1976. Some simplified np-complete graph problems. *Theoretical Computer Science 1*, 237–267.

GOPI, M., AND EPPSTEIN, D. 2004. Single-strip triangulation of manifolds with arbitrary topology. In *EUROGRAPHICS*, 371–379.

GOTTSCHALK, S., LIN, M., AND MANOCHA, D. 1996. OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96*, 171–180.

HEBER, G., BISWAS, R., AND GAO, G. 2000. Self-avoiding walks over adaptive unstructured grids. In *Concurrency: Practice and Experience*, 85–109.

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

ISENBURG, M., AND GUMHOLD, S. 2003. Out-of-core compression for gigantic polygon meshes. In *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, vol. 22, 935–942.

ISENBURG, M., AND LINDSTROM, P. 2004. Streaming meshes. Tech. Rep. UCRL-CONF-201992, LLNL.

KANNAN, R., LOVASZ, L., AND SIMONOVITS, M. 1997. Random walks and an $O(n^5)$ time algorithm for volumes of convex sets. *Random Structures and Algorithms*, 1–50.

KARNI, Z., BOGOMJAKOV, A., AND GOTSMAN, C. 2002. Efficient compression and rendering of multi-resolution meshes. In *IEEE Visualization*, 347–354.

KARYPIS, G., AND KUMAR, V. 1998. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 96–129.

LASSERRE, J. B., AND ZERON, E. S. 2001. A laplace transform algorithm for the volume of a convex polytope. *Journal of the ACM*, 1126–1140.

LIN, M., AND MANOCHA, D. 2003. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*.

LINDSTROM, P., AND PASCUCCI, V. 2001. Visualization of large terrains made easy. *IEEE Visualization*, 363–370.

OLIKER, L., LI, X., HUSBANDS, P., AND BISWAS, R. 2002. Effects of ordering strategies and programming paradigms on sparse matrix computations. In *SIAM Review*, 373–393.

PASCUCCI, V., AND FRANK, R. J. 2001. Global static indexing for real-time exploration of very large regular grids. *Super Computing*, 225–241.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. Qsplat: A multiresolution point rendering system for large meshes. *Proc. of ACM SIGGRAPH*, 343–352.

SAGAN, H. 1994. *Space-Filling Curves*. Springer-Verlag.

SEN, S., CHATTERJEE, S., AND DUMIR, N. 2002. Towards a theory of cache-efficient algorithms. *Journal of the ACM 49*, 828–858.

SILVA, C., CHIANG, Y.-J., CORREA, W., EL-SANA, J., AND LINDSTROM, P. 2002. Out-of-core algorithms for scientific visualization and computer graphics. *Visualization'02 Course Notes*.

VAN KREVELD, M., VAN OOSTRUM, R., BAJAJ, C., PASCUCCI, V., AND SCHIKORE, D. R. 1997. Contour trees and small seed sets for isosurface traversal. In *Proceedings of the 13th International Annual Symposium on Computational Geometry (SCG-97)*, ACM Press, New York, 212–220.

VELHO, L., AND GOMES, J. D. 1991. Digital halftoning with space filling curves. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, T. W. Sederberg, Ed., vol. 25, 81–90.

VITTER, J. 2001. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 209–271.

WILSON, A., LARSEN, E., MANOCHA, D., AND LIN, M. C. 1999. Partitioning and handling massive models for interactive collision detection. *Computer Graphics Forum (Proc. of Eurographics) 18*, 3, 319–329.

YOON, S.-E., AND MANOCHA, D. 2005. Cache-Oblivious Layouts of Bounding Volume Hierarchies. Tech. rep., University of North Carolina-Chapel Hill.

YOON, S., SALOMON, B., LIN, M. C., AND MANOCHA, D. 2004. Fast collision detection between massive models using dynamic simplification. *Eurographics Symposium on Geometry Processing*, 136–146.

YOON, S.-E., SALOMON, B., GAYLE, R., AND MANOCHA, D. 2004. Quick-VDR: Interactive View-dependent Rendering of Massive Models. *IEEE Visualization*, 131–138.