EUROGRAPHICS '2000 / M. Gross and F.R.A. Hopgood
(Guest Editors)

Volume 23, (*2004*), *Number 3*

External Memory View-Dependent Simplification

Jihad El-Sana
Department of Computer Science
Ben-Gurion University
Beer-Sheva, 84105
Israel

Yi-Jen Chiang
Department of Computer and Information Science
Polytechnic University
Brooklyn, NY 11201
USA

Abstract

In this paper, we propose a novel external-memory *algorithm to support view-dependent simplification for datasets much larger than main memory. In the preprocessing phase, we use a new* spanned sub-meshes simplification *technique to build view-dependence trees I/O-efficiently, which preserves the correct edge collapsing order and thus assures the run-time image quality. We further process the resulting view-dependence trees to build the* meta-node *trees, which can facilitate the run-time level-of-detail rendering and is kept* in disk. *During run-time navigation, we keep in main memory only the portions of the meta-node trees that are necessary to render the current level of details, plus some* prefetched *portions that are likely to be needed in the near future. The prefetching prediction takes advantage of the nature of the run-time traversal of the meta-node trees, and is both simple and accurate. We also employ the implicit dependencies for preventing incorrect foldovers, as well as main-memory buffer management and parallel processes scheme to separate the disk accesses from the navigation operations, all in an integrated manner. The experiments show that our approach scales well with respect to the main memory size available, with encouraging preprocessing and run-time rendering speeds and without sacrificing the image quality.*

1. Introduction

Recent advances in three-dimensional acquisition, simulation, and design technologies have led to generation of datasets that exceeds the main memory size and the interactive rendering capabilities of current graphics hardware. Several software and algorithmic solutions have been proposed to bridge the increasing gap between hardware capabilities and the complexity of the graphics datasets. These include level-of-detail rendering with multi-resolution hierarchies, occlusion culling, and image-based rendering.

Recently, view-dependent simplifications have been introduced to enable fine-grained changes to multiresolution hierarchies that depend on parameters such as view location, illumination, and speed of motion. Such simplifications change the mesh structure at every frame to adapt to just the right level of detail necessary to faithfully represent the visual realism. Current such schemes, however, usually increase the size of the dataset, and require the existing of the entire dataset in main memory. This, unfortunately, is a serious drawback, because it limits the applicability of these simplification approaches to only those datasets that do not exceed the main memory size, or otherwise there will be a lot of page faults during both the preprocessing phase and the user navigation phase, resulting in a major slow-down in both phases and, in particular, no interactive navigation performance can be achieved.

In this paper, we propose an *external-memory* (or *out-of-core*) technique to efficiently support view-dependent simplification for datasets much larger than main memory. Our approach is a novel extension of the (binary) *view-dependence trees* of [9], which originally was entirely kept in main memory to facilitate the run-time level-of-detail rendering, and was constructed with the entire dataset kept in main memory. Our new preprocessing algorithm places the dataset *in disk*, and constructs view-dependence trees I/O-efficiently. This is based on a novel, I/O-efficient *spanned sub-meshes simplification* technique. We then further process the view-dependence trees to construct the *meta-node trees*, which in some sense are B-tree-like, to facilitate I/O-efficient traversal. During run-time navigation, we always keep the entire meta-node trees *in disk*, and keep in main memory *only* those *active* meta-nodes that are necessary to

© The Eurographics Association and Blackwell Publishers 2004.Published by Blackwell Publishers, 108 Cowley Road, Oxford OX4 1JF, UK and 350 Main Street, Malden, MA 02148, USA.

render the current level of detail, plus some *prefetched* meta-nodes that are likely to be needed in the near future. Taking advantage of the spatial coherence of the view location, the prefetching prediction is guaranteed to be accurate by the nature of the run-time traversal of the meta-node trees.

We remark that Funkhouser *et al.* [11] also used some prefetching technique for interactive walk-throughs in large architectural virtual environments. Their prefetching method, however, uses the special property of the architectural models that the viewer at any time is in some room and thus only that room together with some small portion of the model visible from the viewer needs to be rendered. Prefetching is carried out by first prefetching the (immediate) neighboring rooms, the rooms neighboring the immediate neighbors, and so on, based on the shortest distance to the viewer. While their technique is restricted to the case of architectural models, our approach is more general and is not subject to such restriction.

As for out-of-core preprocessing method for view-dependent simplification, we remark that Hoppe [17] proposed a method specialized for terrain rendering, by partitioning surface geometry into blocks and using bottom-up recursion to simplify and merge the block geometries. While this works well for terrain datasets, for general 3D datasets, it does not comply with the usual simplification-based scheme in which we collapse edges from the shortest to the longest (with respect to a given *simplification metric* such as Euclidean distance and quadric error metrics [12]), because the block-boundary edges are collapsed *after* the interior edges of the block, resulting in the possibility of collapsing shorter edges *too late* (i.e., if the boundary edges are shorter) and thus likely to cause visual artifacts during navigation. Our spanned sub-meshes simplification technique, on the other hand, guarantees that the edge collapses are always performed in the correct order, and moreover in an I/O-efficient way.

Several additional ideas are used in our method, including the use of *implicit dependency* developed in [9] (for preventing undesirable foldovers) which requires only *local* accesses of information and is especially amiable for the external-memory approach. We also employ our own main-memory buffer management for allocating/flushing place holders in main memory for the meta-nodes of the meta-node trees during run-time navigation. In addition, two processes are used during run-time, one in charge of the navigation operations, the other in charge of the disk prefetching and the main-memory buffer management, so that the overhead of the external-memory support to the navigation performance is minimized. As with the view-dependence trees of [9], our technique supports geometry as well as topology simplification, and handles non-manifold cases.

With our algorithm, we achieve navigation rendering speed 4.4–4.73 times as fast as the state-of-the-art main-memory view-dependent rendering algorithm whose underlying data structure cannot fit in main memory, with a similar image quality. For some situations, we even achieve an improvement from "not being able to navigate" to 4.5–5.6 average frames per second.

2. Previous Work

In this section we give an overview of previous work done in the areas of view-dependent simplifications and external-memory techniques.

2.1. View-Dependent Simplifications

Most of the previous work on generating multiresolution hierarchies for level-of-detail-based rendering has concentrated on computing a fixed set of view-independent levels of detail. At runtime an appropriate level of detail is selected based on viewing parameters. Such methods are overly restrictive and do not take into account finer image-space feedback such as light position, visual acuity, silhouettes, and view direction. Recent advances to address some of these issues in a view-dependent manner take advantage of the temporal coherence to adaptively refine or simplify the polygonal environment from one frame to the next. In particular, adaptive levels of detail have been used in terrains by Gross *et al* [13] and Lindstrom *et al* [19]. Gross *et al* define wavelet space filters that allow changes to the quality of the surface approximations in locally-defined regions. Lindstrom *et al* define a quadtree-based block data structure that provides a continuous level of detail representation. In these approaches, the level of detail around any region can adaptively refine in real-time. These lines of research provide elegant solutions for terrains and other datasets that are defined on a grid. Most of the work for view-dependent simplifications for general polygonal models is closely related to the concept of progressive meshes that are summarized next.

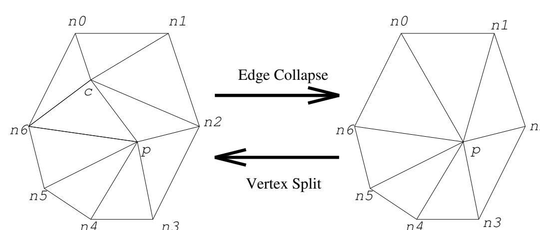


Figure 1: *Edge collapse and vertex split*

Progressive meshes have been introduced by Hoppe [15] to provide a continuous resolution representation of polygonal meshes. Progressive meshes are based upon two fundamental operators – edge collapse and its dual, the vertex split, as shown in Figure 1. A polygonal mesh $\hat{M} = M^k$ is simplified into successively coarser meshes M^i by applying a sequence of edge collapses. One can retrieve the successively higher detail meshes from the simplest mesh M^0 by

© The Eurographics Association and Blackwell Publishers 2004.

using a sequence of vertex-split transformations. The sequence $(M^0, \{split_0, split_1, \ldots, split_{k-1}\})$ is referred to as a *progressive mesh* representation.

Merge trees have been introduced by Xia *et al* [27] as a data structure built upon progressive meshes to enable real-time view-dependent rendering of an object. These trees encode the vertex splits and edge collapses for an object in a hierarchical manner. Hoppe [16] has independently developed a view-dependent simplification algorithm that works with progressive meshes. This algorithm uses the Screen-space projection and orientation of the polygons to guide the run-time view-dependent simplifications. Luebke and Erikson [20] define a *tight octree* over the vertices of the given model to generate hierarchical view-dependent simplifications. If the screen-space projection of a given cell of an octree is too small, all the vertices in that cell are collapsed to one vertex. Gueziec *et al* [14] demonstrate a surface partition scheme for a progressive encoding scheme for surfaces in the form of a directed acyclic graph (DAG). Klein *et al* [18] have developed an illumination-dependent refinement algorithm for multiresolution meshes. Schilling and Klein [23] have introduced a refinement algorithm that is texture dependent. El-Sana *et al* [8] have developed Skip Strip: a data-structure that efficiently maintains triangle strips during view-dependent rendering.

2.1.1. View-Dependence Tree

View-dependence tree was introduced by El-Sana and Varshney [9], then they introduce a parallel construction algorithm for view-dependence trees [10] to reduce the preprocessing time in multi-processors machines. Since our technique extends the view-dependence tree, we review this structure here in more detail. This tree differs from other previous work [27,16] in that it enables topology simplification, does not store explicit dependencies, and handles non-manifold cases. At run-time the view-dependence tree is used to guide the selection of the appropriate level of detail based on factors such as view and illumination parameters.

To enable topology simplification, a pair of vertices that are not connected by an edge are allowed to collapse. This will allow merging of unconnected components. Such a vertex pair is said to be connected by a *virtual edge*, while the original model edges are referred to as *real edges*. To generate the virtual edges, they compute the 3D Voronoi diagram whose sites are the dataset vertices, and connect every pair of vertices by a virtual edge if they are not connected via a real edge and their corresponding Voronoi cells share a Voronoi face.

To be able to handle non-manifold cases, a more general scheme is used so that when a vertex split occurs, more than two new adjacent triangles can be added that share the newly created edge (in the case of a manifold each edge is shared by no more than two triangles). The use of implicit dependencies to prevent undesirable foldovers is discussed in Section 3.3.5.

2.2. External Memory Techniques

We now briefly review the work on external-memory techniques. In addition to early work on sorting and scientific computing, recently there have been external-memory algorithms for graphs and for computational geometry; see [3,5] for the references. Although most of the results are theoretical, the experiments of Chiang [2], Vengroff and Vitter [26], and Arge *et al.* [1] on some of these techniques show that they result in significant improvements over traditional algorithms in practice. Teller *et al.* [24] describe a system to compute radiosity solutions for polygonal environments larger than main memory, and Funkhouser *et al.* [11] present prefetching techniques for interactive walk-throughs in large architectural virtual environments. More recently, Pharr *et al.* [21] give memory-coherent ray-tracing algorithms, Cox and Ellsworth [7] present application-controlled demand paging methods, and Ueng *el al.* [25] propose out-of-core streamline techniques. Also, Chiang and Silva [3,4] and Chiang *et. al* [5] give a series of external-memory approaches for isosurface extraction from volumetric datasets. As mentioned before, Hoppe [17] proposes view-dependent simplification method based on surface geometry blocking for terrains larger than main memory.

3. Our Approach

Our approach consists of two phases: an off-line preprocessing phase, and an on-line navigation phase. In the off-line preprocessing phase, we construct the view-dependence trees using our I/O-efficient *spanned sub-meshes simplification* technique, and build the *meta-node trees*, for the given dataset that cannot fit in main memory. We keep the result of this phase, the meta-node trees, in disk. In the on-line navigation phase, the meta-node trees are used to facilitate the run-time navigation through the given dataset.

3.1. I/O-Efficient View-Dependence Trees Construction

In this section, we develop our *spanned sub-meshes simplification* technique for constructing the view-dependence trees I/O-efficiently.

The original view-dependence trees [9] are constructed bottom-up by recursively collapsing edges (real and virtual edges) in shortest-first order. Notice that this shortest-first order is with respect to a given simplification metric, such as Euclidean distance and quadric error metrics [12]. Since each collapsed edge determines the *switch value* of the newly generated node (the parent), and the switch values of the nodes influence the refinement process at run time, it is very important to preserve the correct shortest-first order of collapsing edges in order to ensure the rendering quality at run time.

To construct view-dependence trees for a dataset larger than main memory, we can only simplify some portion of the dataset mesh at a time by loading that portion into main memory. While such sub-mesh is being simplified, the corresponding view-dependence (sub-)trees are constructed at the

© The Eurographics Association and Blackwell Publishers 2004.

same time. The major challenge is that we want to simplify as much as possible for each main memory load to reduce the amount of I/O operations, while preserving the correct order of collapsing edges.

3.1.1. Sub-meshes Generation

Intuitively, we would like to partition the dataset mesh into disjoint sub-meshes $m_0, m_1, m_2, ..., m_k$, and simplify them independently. But observe that there is another "leading force" in the view-dependence trees construction: visiting edges from the shortest to the longest, to collapse edges in that order. Our algorithm, the *spanned sub-meshes simplification* technique, combines the two ideas together, by introducing the concept of the *spanning subgraph* of a sub-mesh. We use spanning subgraphs to obtain the sub-meshes to be simplified, and during the simplification of a sub-mesh, we make sure that the neighboring edges incident to the sub-mesh are all no shorter than the edges of the sub-mesh that are collapsed.

Before we discuss the actual algorithm, we first define a *span* relationship between the edges and triangles of a sub-mesh. We say that a triangle t is *spanned* by a set of edges S_e if one of the following holds.

i. One of the edges of the triangle t belongs to the set S_e.
ii. The three vertices of the triangle t are also vertices of some edges that belong to the set S_e.

We say that a set of edges S_e *spans* a sub-mesh m if all the triangles of m are spanned by S_e. For convenience, we call the *largest* such sub-mesh M the *sub-mesh spanned by* S_e. The set S_e is called the *spanning sub-graph* of M. Notice that since M is connected, the edges in S_e form a connected graph. We call the neighboring edges of M that are incident to M but are not part of M the *exterior boundary edges* of M. Figure 2 shows a mesh in thin lines and its *spanning graph* in bold lines.

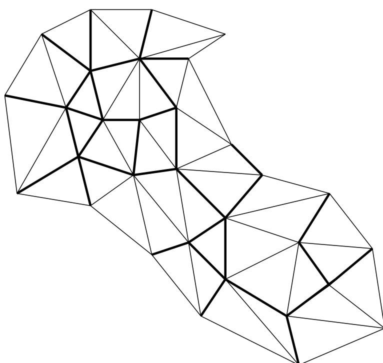


Figure 2: *A mesh with its spanning subgraph (in bold lines).*

The main idea of our algorithm is to include edges, in the shortest-first order, as *spanning edges*, each connected component of which defines a disjoint spanned sub-mesh to be simplified. The spanning sub-graph of each sub-mesh also gives an edge-length upper bound for the edges to be collapsed, to preserve the correct collapsing order, as described in more details next.

3.1.2. The Spanned Sub-meshes Simplification Algorithm

Now we give a full description of our *spanned sub-meshes simplification* algorithm, as follows.

1. Externally sort all edges in the dataset mesh from the shortest to the longest (with respect to a given simplification metric), and store them into a B-tree. Each edge in the B-tree also contains the information about the triangles sharing the edge, and is maintained in the B-tree using the edge length as the key.
2. Delete the edges from the B-tree in the shortest-first order, and load them into main memory as the *spanning edges*. Each connected component of these spanning edges defines a corresponding spanned sub-mesh. Load these spanned sub-meshes into main memory by deleting their edges from the B-tree. As more spanning edges are included, new spanned sub-meshes are created, existing corresponding sub-meshes are grown, or two existing disjoint sub-meshes are merged together if a new spanning edge connects the two sub-meshes (see Fig. 3). Stop this stage of sub-mesh growing when the *sum* of the sizes of the sub-meshes currently in main memory reaches the main memory size. Let ℓ be the longest edge length among all spanning edges included so far.
3. Independently simplify each sub-mesh m currently in main memory, as follows. Collapse the edges of m in the shortest-first order, as usual, by using a (main-memory) priority queue, until all edges with length $\leq \ell$ are collapsed. Build the corresponding view-dependence sub-trees for m as m is being simplified.
4. For each sub-mesh m considered in Step 3., insert the *left-over* edges of the sub-mesh m into the B-tree. This is effectively replacing the *original* sub-mesh m with the *simplified* m into the B-tree. Store the constructed view-dependence sub-trees for m in disk for future use. They can be retrieved later by putting appropriate links to the left-over edges of m.
5. Repeat Steps 2.– 4., until the entire B-tree can fit in main memory, in which case load the entire remaining mesh, i.e., the entire B-tree, into main memory and simplify it.

Consider Step 3., the simplification of each sub-mesh m in main memory. Notice that the *exterior boundary edges* of m are all *no shorter than* ℓ (or otherwise they would have been included into the spanning edges of the current main memory load by our construction), and thus our method of collapsing all edges of m up to edge length ℓ preserves the correct collapsing order, i.e., we never collapse a longer edge before a shorter edge, and thus no visual artifact is introduced. This is one of the most crucial points of the algorithm. At the same time, this step also simplifies the entire

© The Eurographics Association and Blackwell Publishers 2004.

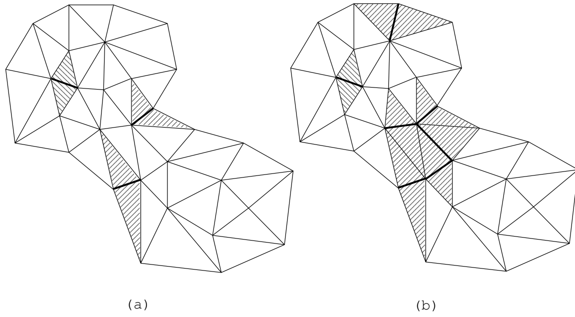


Figure 3: *A set of growing spanning subgraphs generated by the included spanning edges (in bold lines).*

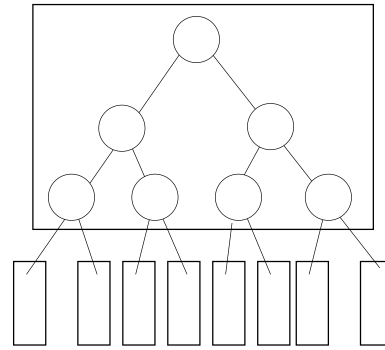


Figure 4: *Meta-node tree $\mathcal{T}$: each circle is a node in the binary view-dependence tree T, and each rectangle, which blocks a subtree of T of L levels (here $L = 3$), is a node of $\mathcal{T}$.*

main memory load sub-meshes as much as possible, resulting in an I/O-efficient computation.

Another crucial point is that in Step 2. we only grow the sub-meshes to the point at which the *sum* of the sizes of the sub-meshes reaches the main memory size. This guarantees that the sub-meshes grown so far all fit in main memory, and thus merging of two sub-meshes can be easily done. If we insisted on finding a sub-mesh as large as main memory size and then simplifying it, as usual partitioning method would do, we would then get into the trouble of not knowing which sub-mesh to keep in main memory and which ones to throw away since we cannot predict which one will grow the fastest. Also, if we were to maintain all sub-meshes in disk, then it would be very difficult to merge two sub-meshes in an I/O-efficient way — for each new spanning edge included, we would have to decide which sub-mesh(es) it is attached to and whether two sub-meshes have to be merged, i.e., we would need to solve the *disjoint sets union-find* problem [6] in external memory, which is still an open problem in the literature of external-memory algorithms. Note that our algorithm handles well the extreme case: when we reach the memory limit with each sub-mesh consists of only two triangles (one edge). In such case our algorithm loads adjacent triangles, performs the test for foldover, and if possible carries out the collapse and updates the adjacent triangles. When the memory is not enough it executes the operation in two stages. It loads each adjacent triangle and tests whether it folds over itself, in case of safe collapse, it then performs the collapse and again loads each triangle and updates its connectivity.

We remark that in our current implementation, the navigation part can support topology simplification, but the preprocessing part cannot actually support it. Recall from Section 2.1.1 that topology simplification requires the construction of virtual edges through 3D Voronoi diagram. At this point, we do not know of any external-memory algorithm for 3D Voronoi diagram. It is possible that we can still generate limited virtual edges, by constructing (in main memory) a 3D Voronoi diagram for each sub-mesh being simplified in Step 3. of the above algorithm. We do not know how well this method can offer, however.

3.2. Meta-Nodes Trees Creation

The view-dependence trees created in the previous section is binary in nature. To facilitate I/O-efficient navigation in the navigation phase, we convert each view-dependence tree T into a *meta-node* tree $\mathcal{T}$, by blocking every subtree of T of L levels, in a top-down fashion, into a *meta-node* (see Figure 4). This is the final stage of the preprocessing algorithm. Here L is a parameter in the program. Every node of the meta-node tree $\mathcal{T}$ is a meta-node, and contains up to $2^L - 1$ vertices (original nodes) of the corresponding view-dependence tree T; the number $2^L - 1$ is achieved if the subtree being blocked is a complete binary tree. We choose L appropriately so that the size of each meta-node roughly matches the disk block size to facilitate efficient disk accesses.

As described in Section 3.3.1, each vertex v (original node) of a view-dependence tree T stores the information about the adjacent triangles of v to obtain the *active* triangles needed for rendering the current level of details. Therefore, each meta-node also contains the information about the adjacent triangles of all the (up to $2^L - 1$) vertices inside this meta-node. We store this information associated with the meta-node in a compact fashion: any triangle that is adjacent to more than one vertex of the meta-node is stored only once, in the *local triangle list* of this meta-node. The adjacent triangle list of v then consists of pointers to the corresponding triangles in the local triangle list of the meta-node. Since we always access the entire meta-node from disk as a whole, this pointer references *within* a meta-node is efficient, while at the same time the compact representation via pointers makes the disk space usage more efficient.

We note that when we collapse a vertex pair, we can position the resulting new vertex at our convenience. In the view

© The Eurographics Association and Blackwell Publishers 2004.

dependence trees, this new vertex is the parent node of the pair of the vertices. If we choose to let the parent use the position of one of its children, say left child, then all internal nodes of the tree use the coordinates of the leaves. We adopt this scheme, and in our meta-node, we only need to store the coordinates and the colors of the leaves of the subtree of the view-dependence tree T blocked into this meta-node. This reduces the necessary space for storing coordinates and colors in a meta-node by a factor of 1/2.

To build meta-node trees, we use a main-memory buffer to hold the meta-node currently being constructed, and traverse the corresponding view-dependence tree using depth-first search for L levels; when the subtree of the view-dependence tree of L levels is entirely visited and the current meta-node is completely constructed, we write the content of the buffer to disk and the buffer is again available for use. The size of the meta-node trees is therefore linear in the size of the corresponding view-dependence trees, and the entire processing time is also linear in the size of the view-dependence trees.

Notice that we use *implicit dependency* developed in [9] (see Section 3.3.5) for preventing foldovers. This requires only *local* accesses of information as opposed to non-local accesses necessary for the use of explicit dependency, and therefore we do not need to block/store the explicit dependency lists in disk. This not only reduces the size of the view-dependence/meta-node trees, but also is especially crucial for our external-memory technique, since non-local accesses in disk is very inefficient and would cause both the design of meta-node trees and the run-time navigation much more difficult.

3.3. Run-Time Navigation

During run-time navigation, our major strategy is to keep the entire meta-node trees *in disk*, and keep in main memory *only* those *active* meta-nodes that are necessary to render the current level of details, plus some *prefetched* meta-nodes that are likely to be needed in the near future. Since the underlying structure of the meta-node trees are view-dependence trees, we first briefly describe how to use the view-dependence trees to perform run-time navigation. We use the term “meta-node” to refer to a node in a meta-node tree, and the term “node” to refer to a node of the original view-dependence tree.

3.3.1. Active Nodes and Active Triangles

For a given input dataset, the view-dependence tree construction often leads to a forest (set of trees) since some nodes can not merge together to form one tree. The view-dependence trees are able to adapt to various levels of detail. Coarse details are associated with nodes that are close to the top of the tree and high details are associated with the nodes that are close to the bottom of the tree. The reconstruction of a real-time adaptive mesh requires the determination of the list of vertices of this adaptive mesh and the list of triangles that connect these vertices, to be sent to the graphics engine for rendering. We refer to these lists as the list of *active nodes* and the list of *active triangles*.

The list of active vertices is a subset of the nodes of the view-dependence trees and is determined by: eye parameters, such as eye position and look-at direction, light parameters, such as position and direction, and distance metric function, which determines the level of details at each vertex.

At each frame the set of active nodes is traversed and for each node we use the distance metric to compute a metric value. This metric value represents the distance to the viewer, the light source, and the local geometry. We then compare the metric value at a node with the *switch value* stored at that node to determine the next operation to execute.

If the metric value is less than the switch value and this node satisfies the implicit dependency conditions for split (to prevent possible foldovers after splitting; see Section 3.3.5), we split this node into its two children. If the computed metric value is larger than the switch value stored at the parent of this node and its sibling can collapse, we collapse this node and its sibling. Otherwise, this node stays in the active nodes list.

The split operation involves removing the node from the active nodes list and inserting its two children into this list. In addition, we need to update the active triangles list, by inserting the newly created adjacent triangles due to this split, which are obtained by looking at the adjacent triangle lists[†] stored in the two children. This is the reason why we need to store the adjacent triangle list for each node, as mentioned in Section 3.2. The collapse operation is an inverse operation, and we update the active nodes list and the active triangles list accordingly.

3.3.2. External Memory Support

Now we describe our navigation approach using external-memory support. From Section 3.3.1, we know that an active meta-node is a meta-node that contains an active node of the view-dependence trees. At any time during navigation, we keep in main memory the active meta-nodes, together with the meta-nodes that are either the parent or the child meta-nodes of the active meta-nodes; these parent/child meta-nodes are *prefetched* for possible future use. (Initially, we load into main memory all the root meta-nodes as the active meta-nodes, and prefetch all their child meta-nodes. The navigation starts with the root vertices of the view-dependence trees, i.e., the least detailed level.) In this

[†] These lists are called *permanent adjacent triangle* (PAT) lists [9] and are different from the ordinary adjacent triangle lists; we omit the details here.

© The Eurographics Association and Blackwell Publishers 2004.

way, as we switch up or down on different levels of the meta-node trees during navigation, we prefetch/flush meta-nodes so that usually three levels of meta-nodes are kept in main memory (see Fig. 5). Notice that switching between levels of the view-dependence trees that are in the same meta-node does not cause any prefetching/flushing.

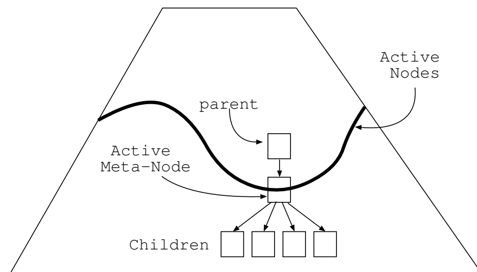


Figure 5: *The meta-nodes of the meta-node trees kept in main memory during run-time navigation.*

Due to spatial coherence of the view location, switching between levels of details always occurs between adjacent levels in the view-dependence trees, and hence the above prefetching prediction would have a 100% hit rate if all prefetching requests were satisfied. However, we may encounter the situation where there is no free main memory for prefetching; in this case we just give up prefetching. It is also possible that we want to switch up or down to some meta-node that are not in main memory due to previous giving up of prefetching; in this case we just render the level of details that exists in main memory and best matches the desired level (namely, we abort switching up or down), and at the same time sending a prefetching command for the missing meta-node for use in the near future.

Now consider the situation where all main memory space is occupied by meta-nodes that are either in use or were prefetched for future use. In this case, we are stuck with a current active meta-node and cannot even switch up to its parent meta-node if that parent meta-node is missing, because the prefetching command for the missing parent meta-node will always be given up due to the lack of free main memory. Certainly this is undesirable. We call this kind of prefetching request an *urgent* prefetching, and will try to fulfill such prefetching by flushing the first found meta-node that is only prefetched but is not being used, to make room for the urgent prefetching. Similarly, we consider the prefetching request for a missing child meta-node from a switch-down attempt as an urgent prefetching.

In addition, we want to avoid the situation in which all main memory space is occupied by meta-nodes that are all *in use*, since in this case we might be again stuck and could not even switch up to parent meta-nodes. To prevent such undesirable situation, we allow urgent prefetching for switching *down* only when there are still at least three meta-nodes in main memory that are *not* in use (but were prefetched). Note that switching down to child meta-nodes that have been urgently prefetched will increase the number of meta-nodes in use by at most two, leaving at least one meta-node not in use. On the other hand, we always allow switch-up urgent prefetching since switching up can only decrease the number of meta-nodes in use. In this way, our main memory is never entirely occupied by meta-nodes that are all in use.

To support the above tasks, as well as efficient main memory allocation/de-allocation, we need a main-memory buffer management scheme.

3.3.3. Main Memory Buffer Management

We first define some terminology. In our scheme, we have a chunk of main memory, actually an array of "place holders", each of size just enough to hold the largest meta-node in disk. Typically meta-node sizes are not the same but do not differ too much. When a meta-node resides in some place holder, it is said to be in the *physical main memory*. We use a hash table to keep track of the meta-nodes currently in the place-holder array. So if a meta-node can be found through the hash table search, then it is in the physical main memory. We also maintain a free list to keep track of all place holders that are free to use. Intuitively, a place holder can only be either in the hash table or in the free list, but not both. But consider the situation in which a meta-node flushed in the previous step is now needed in the next step. To handle this situation more efficiently, when a meta-node is flushed, we still keep its entry in the hash table so that its content is still available, and only add an entry for this place holder in the free list. Then when such a flushed meta-node is requested again, we perform a hash-table search to find that it is in physical main memory, and remove its free list entry, obtaining the meta-node content without an actual disk read. This is called the "second chance" and can greatly improve performance. For a meta-node that is not flushed, it is in the hash table and the corresponding place holder is not in the free list. We say that such meta-node is in the *real main memory*. Therefore, a meta-node is either in disk or in the physical main memory. If it is in the physical main memory, then it can still not be in the real main memory if it is also in the free list. Only when a meta-node is reachable through a hash-table search and also is not in the free list, does it reside in the real main memory.

As part of the start-up step for run-time navigation, we allocate an array of K place holders in main memory as described above, where K is a parameter that can be adjusted according to the available main memory size. For each of such place holder, we maintain two counters: the *usage count*, which records the number of active nodes (of the original view-dependence tree) inside the meta-node M held in this place holder, and *reference count*, which records the number of the active parent meta-node plus the number of the active child meta-nodes of M, i.e., the number of

© The Eurographics Association and Blackwell Publishers 2004.

"prefetching references" to this meta-node *M* from other active meta-nodes. When both counters are 0, the place holder is considered to be a free space for loading a new meta-node from disk, and is flushed by adding its entry to the free list but not removing its entry from the hash table.

As stated, by a hash-table search we know whether a meta-node is in physical main memory, and if so where it is. We also maintain a free list to keep track of the entries of the place-holder array that are currently free to be used, i.e., the place holders whose usage count and reference count are both 0. Each entry in the hash table or the free list is just a pointer (array index) to the place holder. Each place holder has two pointers respectively to its corresponding entries in the hash table and in the free list (null if the place holder has no such entries).

As we switch up or down during navigation, we update the usage counts and the reference counts accordingly. When a place holder holding some meta-node *M* has both counters updated to 0, we flush the meta-node *M*, by putting an entry for this place holder in the free list, indicating that this place holder is free. Notice again that we do not remove the place holder's entry in the hash table, so a hash-table search for *M* can still locate this place holder, to facilitate the second-change scheme. To maximize this second chance, we always put the entry of a newly freed place holder at the *end* of the free list, and always take the free space from the *first* entry of the free list.

When we allocate a place holder for a meta-node *M*, whether it is a "second chance" to bring *M* from physical main memory or it is the case to fetch *M* from disk, we are putting *M* to the *real main memory*, and thus we always need to remove the corresponding free-list entry, otherwise this place holder might be re-allocated to some other meta-node while *M* is still being used. Recall that as we switch up or down during navigation, we update the usage counts and the reference counts accordingly. Attempting to increase the reference count of a meta-node not in real main memory causes a prefetching of that meta-node into real main memory, either from physical main memory or from disk. Attempting to decrease the reference count of a meta-node not in real main memory has not effect.

3.3.4. Parallel Processes Support

One important optimization of our approach is to separate the disk accesses from the run-time navigation, so that the navigation can proceed without waiting for the disk accesses to complete. Our navigation algorithm consists of two parallel processes: *Navigate* and *I/O*. *Navigate* is in charge of the navigation operations, and *I/O* is in charge of the disk prefetching and the main-memory buffer management. The two processes share a command board buffer to which *Navigate* sends commands to be executed and from which *I/O* fetches the commands to execute. The command board buffer is protected by an exclusive lock. The two processes also share the place-holder array, the hash table and the free list, which are protected by another exclusive lock.

Our "second chance" scheme described in Section 3.3.3 makes the locking somewhat tricky. A potential mistake we want to guard against is that a place holder holding an active meta-node used by *Navigate* is considered by *I/O* as a free space and is loaded with some other meta-node. Since the prefetching operations take longer time, *I/O* usually will fall behind *Navigate* and the updates of the reference counts and the usage counts may not reflect the actual counts of the current status. This may cause *I/O* to flush a meta-node (with unupdated counts both being 0) which is currently used by *Navigate*. Therefore we only let *Navigate* maintain the usage count to correctly reflect the current status. Updating the usage count (from *Navigate*) and checking it (from *I/O*) both requires the lock, which can be released shortly. (The reference counts only affect the prefetching hit ratio and do not affect the correctness.) Also, switching up or down in *Navigate* to a meta-node that is in the physical main memory but not in the real main memory will bring the meta-node back to the real main memory with the "second chance". We let *Navigate* hold the lock until the removal of the corresponding free-list entry is done, which again only takes a short time. Similar considerations apply to situations where the free list or the hash table is updated or examined from either process. In each case the lock can be released shortly.

3.3.5. Implicit Dependency

Now we describe the implicit dependency developed in [9] that is used in our method. Dependency checking is necessary to ensure run-time consistency in the generated triangulations. Implicit dependency allows highly localized memory accesses during run-time.

Implicit dependencies rely on the enumeration of vertices generated after each collapse during the construction of the view-dependence trees. If the model has n vertices at the highest level of detail they are assigned vertex-ids $0,1,\ldots,n-1$. Every time a vertex pair is collapsed to generate a new vertex, the id of the new vertex is assigned to be one more than the greatest vertex-id thus far. This process is continued till the entire view-dependence trees have been constructed.

Before split or collapse operation is executed at run-time we make a few simple tests based on vertex ids to ensure the consistency of the generated triangulations and to avoid mesh foldovers. These tests are given as follows. (i) *Vertex-Pair Collapse:* A vertex-pair (a,b) can be collapsed if the vertex-id of their parent is less than the vertex-ids of the parents of the collapsed boundary vertices. (ii) *Vertex Split:* A vertex p can be safely split at runtime if its vertex-id is greater than the vertex-ids of all its neighbors.

In our current implementation of implicit dependencies we store two integers with each view-dependence-tree node

© The Eurographics Association and Blackwell Publishers 2004.

which are (i) the maximum vertex-id of the adjacent vertices and (ii) the minimum vertex-id of the parents of the collapse boundary vertices. The two integers are updated after each change of the collapse boundary as a result of split or collapse. A proof of the correctness of implicit dependencies is given in [9].

We remark that the use of implicit dependencies is crucial to our external-memory approach, since each time we attempt to switch up or down in the view-dependence trees we need to first perform the dependency test to see if such attempt is safe. If we were to use explicit dependencies where the accesses are non-local, such tests would be much more difficult and much less efficient to perform.

4. Results

We have implemented our algorithm in C/C++, tested our non-optimized implementation on several datasets, and received an encouraging results. Part of these results are shown in Tables 1 and 2. The preprocessing time results in Table 1 have been obtained on SGI O2 with 32 MB free RAM before running the program. The run-time results on Table 2 have been obtained on SGI O2 with 80, 96 and 128 MB RAM. For SGI O2 the operating system and other system tools consume about 64-76 MB, therefor we used 16, 24, and 48 MB in our tests, which is the available physical main memory.

Table 1 shows the preprocessing times for constructing the view-dependence trees (*VDT*) and the meta-node trees (*MNT*), the size of the original dataset (*Off*), and the sizes of the generated files (*VDT* for the view-dependence trees, and *DATA+MNT* for the meta-node trees). The numbers of triangles and of vertices of the original datasets are shown as the *Tris* and the *Verts* entries.

As can be seen from Table 1, our meta-node trees construction takes much less time than the construction of the view-dependence trees. We first construct the view-dependence trees (*VDT*) from the original dataset file (*Off*), then we convert the *VDT* file into an I/O-efficient representation of the meta-node trees, stored as a data block file (*DATA*) and a tree-node block file (*MNT*), to allow fast access in disk. The construction times for both the view-dependence trees and the meta-node trees are more or less linear in the size of the dataset regardless of whether it exceeds the main memory size, showing that the algorithms scale well with respect to the main memory size.

Note that the *Off* format is an *ASCII* representation of the dataset while *VDT* is a compact binary representation of the view-dependence trees. The size of the meta-node trees (the *DATA* plus the *MNT* files) is larger than the *VDT* file as a result of our blocking scheme to achieve an I/O-efficient traversal. It is important to note that such extra space is not crucial for our algorithm, for two reasons. First, while the entire *VDT* file has to reside in main memory for the main-memory view-dependence trees algorithm, our algorithm only needs to load a very small portion of the meta-node trees into main memory, and hence is much more amiable to large datasets. Second, compared to the great interactivity improvement during navigation offered by the meta-node trees, the *disk-space* increase by a factor of 2.25 on average is actually very cost-effective.

Table 2 shows the results of the run-time navigation using the view-dependent rendering algorithm that was built on top of our external-memory support, and the same view-dependent rendering algorithm using virtual memory.

It is important to test our system over several frames in order to measure the interactivity, the changes between the consecutive frames, and the performance of the external-memory support system. Therefore, for each dataset we record a path which enforces the same number of frames and the same image quality for each frame when using different memory sizes or different algorithms (for navigation along the same path). Hence, it is enough to measure the frame rate in order to test the performance of our algorithm. One could also keep the frame rate constant and measure the quality of the images. Since it is not easy to measure the quality of the images we chose to use the first method.

In table 2 we use the same path and the same number of frames for each main memory size we test (16, 24, and 48MB). For each case, we allocate in our program as much main memory as available, but if the entire meta-node trees can fit in main memory, we never allocate main memory that is too much larger than necessary. We average the number of vertex splits (switch down) and vertex-pair merges (switch up) over the given path. We refer to this number as the *Adapt* count. *Tris* is the average number of triangles rendered per frame along the given path. Each time the navigator asks for switching up or down but the external-memory support can not fulfill this request we count this as one miss. In Table 2 *Miss* is the percentage of misses per frame (the average misses along the path). *Virt.* is the average rendering time (in seconds) per frame along the path when using virtual memory. *Ext.* is the average rendering time (in seconds) per frame when using our external-memory support. Note that *Virt.* will always have 0% Miss rate since it is always waiting for the page fault to complete, getting the requested information, and then proceeds. In a sense, *Miss* measures the image quality, while *Virt.* and *Ext.* entries measure the interactivity of the algorithms.

Regarding to Table 2, we make the following observations.

- For small datasets and/or large main memory where the entire view-dependence trees can fit in main memory, *Ext.* performs a little worse than *Virt.*. This is expected, since *Ext.* has the extra overhead of main-memory buffer management, etc.. It is interesting to see that although *Ext.* is a little worse, the performance is still comparable to *Virt.*,

© The Eurographics Association and Blackwell Publishers 2004.

Dataset	Number Of		Const. Time(sec)		File size in MB			
	Tris	Verts	VDT	MNT	Off	VDT	DATA	MNT
Bunny	69 K	36 K	9.6	1.8	3.1	2.6	6.3	0.8
Knee	75 K	37 K	10.8	0.3	3.6	2.4	2.8	0.6
Dragon	202 K	101 K	31.7	5.4	6.8	7.8	19.3	2.5
BallJoint	274 K	137 K	38.6	4.7	13.3	13.2	24.1	3.1
Buddha	293 K	145 K	42.3	5.5	13.2	11.3	28.0	3.7
Submarine	339 K	173 K	53.6	6.2	11.8	10.5	26.2	5.1
Terrain	522 K	262 K	71.1	2.5	20.1	16.7	17.5	1.8
Steve	739 K	272 K	105.9	13.1	28.7	27.3	55.4	7.4
David	1,172 K	588 K	213.4	11.8	45.6	42.7	84.8	11.1

Table 1: *Preprocessing times and the sizes of the generated files.*

showing that our main-memory buffer management system is efficient.

- For large datasets and/or small main memory where the view-dependence trees can not fit in main memory, *Ext.* performs much better than *Virt.*, about 4.4–4.73 times as fast. Also, *Ext.* scales quite well with respect to different main memory sizes: the rendering time only increases slightly as the main memory size decreases. This is especially advantageous when *Virt.* cannot run on the dataset Steve with the 16MB main memory configuration, and similarly for the David dataset on both the 16MB and 24MB configurations (the three “N/A” entries in the table). For these cases, while *Ext.* achieves 4.5–5.6 average frames per second, for *Virt.* the OS simply complained that there was no enough swap space and the *Virt.* program could not even start navigation!
- The Miss entries show that *Ext.* have a low miss rate, indicating that our image quality is similar to that obtained from *Virt.* with enough main memory to hold the entire view-dependence trees. Observe that when we have a larger main memory, we can prefetch more meta-nodes, and thus the miss rate is lower, as expected. When the main memory is large enough to fulfill the prefetching requests at any time, the miss rate is 0.

Figures 6*, 7*, and 8* show images generated by our system. Figure 6* shows different resolution (Figure 6*(a) and Figure 6*(b)) of the Dragon dataset. The dynamic changes on the model resolution allow view-dependent rendering at interactive rate (about 6-8 frames/second) for the main memory configuration of only 16MB. Figure 6*(c) shows the wire frame of the low resolution. Figure 7* shows two different level-of-detail representations for the Terrain dataset. Figure 8*(a) shows a selected view in highest detail. Figure 8*(b) shows how we can achieve high level of detail on a selected view by lowering the resolution of regions far from the viewer.

We have also attached to this paper two video segments Dragon.mov and Terrain.mov (in QuickTime format). The Dragon segment shows what the viewer will see on the right top corner window, while the rest of the window shows how the detail changes over the entire model. We generated this segment by merging two segments that we recorded in real-time (separately). Each of these segments runs at about 6-8 frames/second using SGI O2 with about 24 MB free main memory (80 MB total physical memory where 64 MB were used before we started our program). The Terrain segment, which runs at about 6-8 frames/second, was also recorded in real-time on the same machine.

5. Conclusions

We have presented an external-memory technique for view-dependent simplification. For small datasets where the original view-dependence trees can fit in main memory, our algorithm gives the same image quality, performs slightly slower but is still comparable. For large datasets where the view-dependence trees cannot fit, our algorithm performs 4.4–4.73 times as fast, with image quality similar to that of the main-memory view-dependence trees method as if the entire view-dependence trees could fit. For some cases, our algorithm even improves from “not being able to navigate” to 4.5–5.6 average frames per second. Also, our I/O-efficient preprocessing algorithm scales well with respect to the available main memory size.

There are several places that we would like to improve in the future. First, we would like to optimize the navigation part in terms of implementation, which we believe could improve the frame rate of our algorithm.

Second, we would like to incorporate some techniques that anticipate future viewing parameters when making prefetching decisions. For example, by using the current and last few frames, we can compute the trajectory and acceleration of the viewer motion. This information could enable us to predict the viewer position and other viewing parameters in the near future, and therefore facilitate our prefetching tasks.

© The Eurographics Association and Blackwell Publishers 2004.

Dataset	Avrg/frame		16 MB			24 MB			48 MB		
	Adapt	Tris	Virt.	Ext.	Miss	Virt.	Ext.	Miss	Virt.	Ext.	Miss
Bunny	1.1K	22.3K	0.12	0.13	0	0.12	0.13	0	0.12	0.13	0
Knee	1.0K	21.1K	0.12	0.12	0	0.11	0.13	0	0.11	0.13	0
Dragon	1.8K	37.2K	0.24	0.15	0.5	0.16	0.15	0	0.12	0.14	0
BallJoint	1.9K	38.1K	0.31	0.15	1	0.20	0.15	0.1	0.14	0.15	0
Buddha	2.4K	46.2K	0.32	0.15	1	0.21	0.15	0.5	0.14	0.15	0
Submarine	2.7K	53.2K	0.40	0.16	2	0.23	0.15	1.5	0.14	0.15	0.5
Terrain	2.4K	41.2K	0.36	0.15	4	0.23	0.15	2	0.14	0.15	0.5
Steve	3.6K	56.5K	N/A	0.18	7	0.8	0.18	4	0.4	0.17	1
David	5.1K	68.1K	N/A	0.22	12	N/A	0.21	9	0.9	0.19	5

Table 2: *Run-time performance. Note that there are two columns* Adapt *and* Tris *under* Avrg/frame*, and for each of the main memory configurations (*16MB, 24MB*, and* 48MB*) there are three columns* Virt.*,* Ext.*, and* Miss*.*

Finally, the current external-memory algorithm to construct the view-dependence trees does not actually support topology simplification (except for a possible limited support; see the discussions at the end of Section 3.1.2), which is often crucial for large datasets. Current algorithms to simplify topology [22, 12, 9] rely on the condition that the entire dataset fits in main memory. It would be nice to develop an algorithm that can simplify topology efficiently for datasets that exceed the main memory size.

6. Acknowledgements

We would like to thank Amitabh Varshney for many useful discussions about this work. Jihad El-Sana has been supported in part by The Weiler Family Fund. Steve, David, Knee, and BallJoint models have been provided by Cyberware; Bunny, Dragon, and Buddha have been provided by the Stanford Computer Graphics Laboratory. We would like to thank the reviewers for their insightful comments which led to several improvements in the presentation of this paper.

References

1. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 1998.

2. Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. *Comput. Geom.: Theory and Appl.*, 9(4):211–236, 1998.

3. Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *Proc. IEEE Visualization*, pages 293–300, 1997.

4. Y.-J. Chiang and C. T. Silva. Isosurface extraction in large scientific visualization applications using the I/O-filter technique. Technical Report, University at Stony Brook, 1997.

5. Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *Proc. IEEE Visualization*, pages 167–174, 1998.

6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., 1990.

7. M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proc. IEEE Visualization*, pages 235–244, 1997.

8. J. El-Sana, E. Azanli, and A. Varshney. Skip strips: Maintaining triangle strips for view-dependent rendering. In *IEEE Visualization '99 Proceedings*. ACM/SIGGRAPH Press, October 1999.

9. J. El-Sana and A. Varshney. Generalized view-dependent simplification. In *Proceedings Eurographics*, 1999.

10. J. El-Sana and A. Varshney. Parallel construction and navigation of view-dependent virtual environments. In *Conference on Visual Data Exploration and Analysis*, pages 62–70. SPIE Press, January 1999.

11. T. A. Funkhouser, S. Teller, C. H. Séquin, and D. Khorramabadi. Database management for models larger than main memory. In *Interactive Walkthrough of Large Geometric Databases, Course Notes 32, SIGGRAPH '95*, pages E15–E60, 1995. Appeared as "The UC Berkeley System for Interactive Visualization of Large Architectural Models", in *Presence: Teleoperators and Virtual Environments*, Vol.5, No.1, Winter 1996.

12. M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, pages 209 – 216. ACM SIGGRAPH, ACM Press, August 1997.

© The Eurographics Association and Blackwell Publishers 2004.

13. M. H. Gross, R. Gatti, and O. Staadt. Fast multiresolution surface meshing. In G. M. Nielson and D. Silver, editors, *IEEE Visualization '95 Proceedings*, pages 135–142, 1995.

14. A. Gueziec, F. Lazarus, G. Taubin, and W. Horn. Surface partitions for progressive transmission and display, and dynamic simplification of polygonal surfaces. In *Proceedings VRML 98, Monterey, California, February 16–19*, pages 25–32, 1998.

15. H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH '96*, pages 99 – 108. ACM SIGGRAPH, ACM Press, August 1996.

16. H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH '97*, pages 189 – 197. ACM SIGGRAPH, ACM Press, August 1997.

17. H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings IEEE Visualization*, pages 35–42, 1998.

18. R. Klein, A. Schilling, and W. Straßer. Illumination dependent refinement of multiresolution meshes. In *Computer Graphics International*, June 1998.

19. P. Lindstrom, D. Koller, W. Ribarsky, L. Hughes, N. Faust, and G. Turner. Real-Time, continuous level of detail rendering of height fields. In *SIGGRAPH 96 Conference Proceedings*, pages 109–118. ACM SIGGRAPH, August 1996.

20. D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of SIGGRAPH '97*, pages 198 – 208. ACM SIGGRAPH, ACM Press, August 1997.

21. Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In Turner Whitted, editor, *Proceedings of SIGGRAPH '97*, Annual Conference Series, pages 101–108, August 1997.

22. J. Popović and H. Hoppe. Progressive simplicial complexes. In *Proceedings of SIGGRAPH '97*, pages 217 – 224. ACM SIGGRAPH, ACM Press, August 1997.

23. A. Schilling and R. Klein. Texture-dependent refinement for multiresolution models. In *Computer Graphics International*, June 1998.

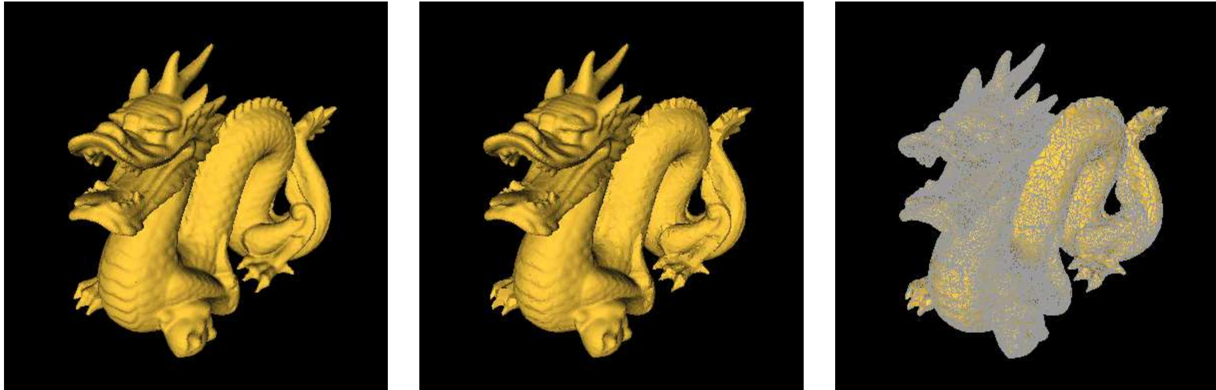
24. S. Teller, C. Fowler, T. Funkhouser, and P. Hanrahan. Partitioning and ordering large radiosity computations. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 443–450. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

25. S. K. Ueng, K. Sikorski, and K.-L. Ma. Out-of-core streamline visualization on large unstructed meshes. *IEEE Transations on Visualization and Computer Graphics*, 3(4):370–380, 1997.

26. D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, 1995.

27. J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, pages 171 – 183, June 1997.

© The Eurographics Association and Blackwell Publishers 2004.

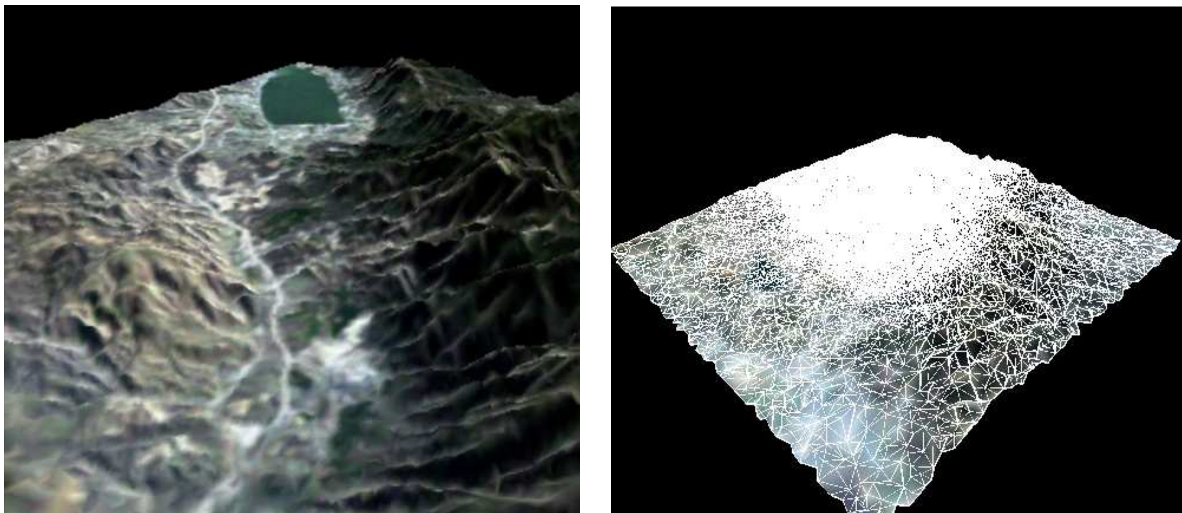


(a) Highest detail:202K tris (b) Low resolution:30K tris (c) Wire frame of (b)

Figure 6: *View-dependent simplification allows interactive navigation for 16MB RAM*

(a) Highest detail:522K tris (b) Low resolution:45K tris (c) Wire frame of (b)

Figure 7: *Uniform low resolution of the terrain that allows interactive navigation for 24MB RAM*



(a) Select view (b) The entire model with the selecetd view

Figure 8: *Highest detail can be achieved in selected view by keeping the rest of the model in very low resolution*

© The Eurographics Association and Blackwell Publishers 2004.