A Multiresolution Representation for Massive Meshes

Eric Shaffer, *Student Member*, *IEEE*, and Michael Garland

Abstract—We present a new external memory multiresolution surface representation for massive polygonal meshes. Previous methods for building such data structures have relied on resampled surface data or employed memory intensive construction algorithms that do not scale well. Our proposed representation combines efficient access to sampled surface data with access to the original surface. The construction algorithm for the surface representation exhibits memory requirements that are insensitive to the size of the input mesh, allowing it to process meshes containing hundreds of millions of polygons. The multiresolution nature of the surface representation has allowed us to develop efficient algorithms for view-dependent rendering, approximate collision detection, and adaptive simplification of massive meshes. The empirical performance of these algorithms demonstrates that the underlying data structure is a powerful and flexible tool for operating on massive geometric data.

Index Terms—Hierarchical data structures, level of detail, mesh simplification, out-of-core algorithms.

1 Introduction

Advances in disk storage, processing power, and laser range scanning have made it possible to produce and store massive polygonal meshes with relative ease. These meshes have polygon counts reaching into the hundreds of millions and typically far exceed the size of physical memory and can even exceed the size of virtual memory. Many traditional data structures and algorithms used in computer graphics and visualization tasks rely on random access to the data. When forced to access significant amounts of data on secondary storage, these methods become inefficient.

As a result, a significant amount of recent research has focused on designing algorithms and data structures to work with meshes that are stored out of core. In this paper, we present a new surface representation for massive polygonal meshes that is distinguished from previous work in the combination of attributes it offers. These include efficient runtime access to a multiresolution representation of the mesh, access to the original surface data, and a memory efficient construction phase. This new data structure combines several well-known ideas, such as octree decomposition of space and vertex clustering, along with innovations such as a new mesh indexing scheme and a scalable construction algorithm based on sorting. One of the contributions of this paper is detailing how these components can be combined to create a flexible and scalable data structure that is relatively easy to implement.

Our surface representation is built as an external memory octree. The bottom level of this octree is constructed by inserting the elements of the mesh into a fine uniform grid. These grid cells link to an indexed mesh representation of the original surface, which forms the finest level of resolution. The octree is completed by a bottom-up merging process, with higher octree levels encoding coarser surface approximations. The construction phase requires only two passes over the input mesh plus external sorts of the vertices and faces. The sorts ensure coherent access of the processed mesh data from disk. Memory usage is proportional to the occupancy rate of the grid and is insensitive to the size of the input mesh. The power and flexibility of this data structure is demonstrated by its use in three algorithms we have developed: a view-dependent renderer, an approximate collision detection system, and a surface simplification tool. The multiresolution nature of the data structure enables these algorithms to work efficiently on massive geometric models.

• *The authors are with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801.*
E-mail: {shaffer1, garland}@cs.uiuc.edu.

Manuscript received 9 Oct. 2003; revised 2 June 2004; accepted 9 Aug. 2004. For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-0099-1003.

2 Related Work

In this section, we review previous work, focusing on only those contributions most relevant to this paper.

2.1 Out-Of-Core Simplification

Several authors, Hoppe [12] and Prince [21] in particular, have proposed methods in which a mesh is segmented so each piece can be simplified in core. Simplification in this setting is usually accomplished via iterative edge contraction [10]. While this method generates high quality results, it can produce long running times. Lindstrom [16] proposed accumulating quadric error information [7] with a uniform grid and simplifying via vertex clustering. This method exhibits memory usage independent of input size and was later extended by Lindstrom and Silva [18] to be completely out of core. Shaffer and Garland [22], [8] proposed a pair of multiphase simplification techniques, processing the original mesh using a uniform grid, and piping the output to an adaptive clustering phase to generate a final approximation. Recently, streaming out-of-core simplification methods have been proposed by Isenburg et al. [13] and Wu and Kobbelt [24]. In both cases, constraints placed on the input format allow the mesh to be processed using a limited amount of in-core storage.

2.2 View-Dependent Rendering

One of the earliest view-dependent rendering (VDR) systems was that of Xia and Varshney [25]. It uses an edge collapse technique to construct a vertex split hierarchy as a preprocessing phase. At runtime, a cut through this hierarchy is determined by the current viewing parameters. Luebke and Erikson [19] generalized this approach to use vertex clusters instead of vertex pair contractions. Hoppe [11] constructed a VDR system based upon his progressive mesh data structure. All of these techniques construct the vertex hierarchy in core.

Several authors have worked on moving VDR out of core. The simplification algorithms of Hoppe [12] and Prince [21] were both used for view-dependent rendering. The former works on terrains while the latter method can handle arbitrary polygonal models. El-Sana and Chiang [6] described a method based on mesh segmentation that achieves interactive frame rates for large models, but requires significant preprocessing time.

DeCoro and Pajarola [5] extended FastMesh [20] to render out of core using a customized paging algorithm. Rendering quality is high, but it is unclear how well the method scales; the data structure is constructed in core, resulting in significant preprocessing times. Borgo et al. [2] created a system for visualizing large meshes which uses a small number of static levels-of-detail generated from the OEMM data structure [4]. The system exhibits good runtime performance, but no timings are given for the construction process. In Guthe et al. [9], an out-of-core view-dependent rendering system is based on an octree decomposition of the mesh. The mesh patch in each cell is simplified and a hierarchical level-of-detail (HLOD) is generated by bottom-up merging. The runtime frame rates reported are excellent. The full comparative performance of the system is unclear as no specific data regarding triangle throughput or HLOD construction time are given.

Lindstrom [17] described a VDR system with an underlying data structure similar to the one we propose; both build out-of-core octrees using vertex clustering and sorting. However, Lindstrom's system employs an external simplification algorithm as a first phase, so, in general, the original mesh will no longer be available. In contrast, our data structure offers explicit access to the original surface. Lindstrom's method does have the benefit of working completely out of core, whereas ours uses some in-core memory to generate the multiresolution structure. More recently, Cignoni et al. [3] developed a VDR system for massive meshes using a tetrahedral decomposition of space to generate the HLOD structure. Like the system we describe in this paper, refinement and coarsening occur of the basis of cells, batching vertex splits, and contractions for the sake of efficiency.

2.3 Collision Detection

While many methods exist for performing collision detection on meshes, few are designed to operate on massive models. The significant work in this area is that of Wilson et al. [23]. Their system performs well, handling interactive collision detection in a 15 million polygon scene using 160 MB of memory. However, no data are given regarding the preprocessing time.

2.4 Mesh Representations

Cignoni et al. [4] proposed an octree-based external mesh representation called OEMM. It offers transparent access to the underlying mesh data, enabling operations such as simplification and mesh editing. In contrast to our octree, which is read-only, the OEMM structure allows read and write access to the mesh. Isenburg et al. [13] proposed a sequenced representation for massive meshes. Here, the mesh is encoded in a streaming format. Mesh accesses are read-only and strictly sequential; decoding a piece of mesh data requires all preceding portions of the mesh be read in first. Neither mesh format includes a multiresolution surface representation. As such, they are suited for a different set of applications than our proposed data structure.

Among all of these contributions, the combination of a scalable construction algorithm, efficient runtime access to a multiresolution representation, and access to the original surface remains elusive. The goal of our work is to fill that gap.

3 External Memory Octree

The central component of our surface representation is an external memory octree. The octree encodes multiple levels of detail, with cells in the bottom level merging to form coarser approximations of the surface in the higher levels. In the following sections, we describe the octree structure in greater detail and discuss how it is constructed.

3.1 Data Structures

In the finest level of the octree, a uniform grid is used to sample the mesh. The dimensions of the grid are $2^n \times 2^n \times 2^n$, where n is chosen by the user. Allocating a full grid is infeasible for large n. Instead, we employ a hash table with 32-bit integer keys. We refer to such a key as an *octindex*. Hashing a vertex into the grid is accomplished by locating the integer grid coordinate for each vertex coordinate and then packing the integer coordinates into a hash key. This constrains each grid axis to employ at most 10 bits, which limits the finest grid resolution to be $1,024^3$. This bounds the quantization error at $1/1,024$ of the length of each axis in the mesh bounding box. For the David statue, with actual dimensions in centimeters of approximately $517 \times 75 \times 57$, the maximum error is then $5 \times 0.7 \times 0.6$ millimeters. This approaches the raw scanning resolution of 0.29 mm along the X and Y axes [14]. The jump from the finest approximate surface data to the actual surface data is not too great in this case, indicating the octree resolution should be acceptable for many meshes.mpling the mesh using a grid results in memory usage insensitive to the size of the input mesh. The memory consumed by the grid is instead proportional to the grid occupancy rate, which is related to the Hausdorff dimension of the surface. We are generally interested in working with heavily sampled range-scan data. Table 3 shows that, for these types of surfaces, the occupancy rate of the octree remains small across a range of mesh sizes. This demonstrates the ability of the data structure to capture reasonable detail in a space efficient manner. Our surface representation is less suitable for large meshes with very high depth complexity, such as space-filling surfaces, where the occupancy rate will be much higher. Large sparsely sampled surfaces, such as massive CAD/CAM models,

TABLE 1
Data Stored in an External Octree Node

float[**3**]	v	proxy vertex position
float[**3**]	$\vec{n}$	normal vector
float	na	normal cone half-angle
float	e	quadric error
integer	o	octindex, packed index of in-core node
integer	f_i	index into face file
short	f_n	number of faces
integer	c_i	index of first child or, if leaf, vertex
short	c_n	number of children or, if leaf, vertices
short	l	level in octree

TABLE 2
Data Stored in an Internal Octreee Map Node

integer	I	linear index of external memory node
integer	a	counter
pointer	p	parent pointer
byte	c	child vector, bits represent a possible child

will also suffer as too few grid cells will be occupied to generate a reasonable approximation of the geometry at lower resolutions. These limitations are not specific to our data structure, but are, in fact, common to all clustering-based methods.

Several types of information are accumulated in the grid during preprocessing. We track the average vertex position within a cell and also gather information related to the faces of the mesh. A face is hashed to the cells associated with its three vertices. Each cell accumulates area-weighted normal vectors and *quadric error* information. A quadric matrix is a 4×4 symmetric matrix that can be used to compute an area-weighted sum of squared distances to a set of planes [7], in this case, the planes of the mesh faces. Quadric matrices are additive and can be stored using only 10 coefficients; both are key properties for our purpose. The quadric matrices for each cell are kept to compute a proxy vertex that minimizes the quadric error function for the octree cell. In the final output of the construction process, this vertex position is saved and the quadric is discarded.

The construction phase generates an octree file consisting of nodes, shown in Table 1, laid out in breadth first order on disk, with sibling cells grouped together. Each node requires 50 bytes of storage. Two temporary data structures are also built: a file of quadrics that is discarded when construction completes and an in-core *octree map*. The octree map contains a 16 byte shadow node, shown in Table 2, for each external octree node and enables efficient hashing into the external memory octree. As this is the only significant in-core data structure, the minimum memory required to process even very large meshes remains modest (typically less then 200MB, see Table 3).

Two other files are included in the final output. These are the vertex file and face file, which together contain the original surface. They are sorted into sets of vertices and faces, respectively, which are associated with specific cells in the octree. Each octree cell in the bottom level has an index into the vertex file and a count indicating how many vertices are contained in the cell. The values in the vertex file record the difference between the vertex in the original mesh and the proxy vertex in the associated octree cell. To save space, these difference vectors are quantized using 16 bits for each coordinate and stored as short integers. This results in a maximal precision of 26 bits per coordinate which, after the mesh bounding box is fixed in space, should be sufficient to capture the uniform precision present in the original floating point value.

Each cell in the octree has an index and range referencing the face file. A face is associated with the cell which contains two or more of its vertices, but whose children contain no more than one. In this cell, the face degenerates into a point or line. When that cell is split, the triangle becomes nondegenerate. For interior nodes, the faces are represented as three octindices, indicating the cells in the bottom grid level which contain the vertices. Each of these indices has an associated offset, which allows the coordinates of the vertex to be recovered from the vertex file. We use a different format for faces associated with leaf nodes. For these faces, at least two of the vertices have octindices identical to that of the leaf node. These identical octindices need not be stored. Instead, only offset fields are stored. Each face uses a 2 byte field to indicate which vertices are represented by offsets alone. This saves from 6 to 10 bytes per face. The relationship between the internal and external structures in the final output can be seen in Fig. 1.

3.2 Construction Algorithm

The input to the octree construction algorithm can be in the form of a polygon soup or an indexed face mesh. Generally, the polygon soup format has been preferred for large data. An indexed face mesh representation potentially requires extra disk reads and seeks if the vertex access pattern does not exhibit locality when the faces are dereferenced. While we have not attempted to measure the coherence of the meshes in Table 3, our experience is that the indexed representations provide acceptable performance. In fact, in our experiments, the polygon soup format performs slightly

TABLE 3
Octree Construction Performance, Disk and Memory Usage in MB, Time in H:M:S Format

Mesh	Input Mesh		Octree Output					
	Triangles	Disk	Nodes	Occupancy	Disk (Peak)	Memory (Peak)	Total Time	IO Time
Buddha	1,087,716	19	757,792	NA	78 (200)	34 (190)	19	4
David 2mm	8,254,150	148	4,512,137	0.3%	421 (929)	82 (810)	7:41	5:56
Lucy	28,055,742	505	4,901,459	0.4%	923 (1,658)	93 (905)	32:06	28:01
David 1mm	56,230,343	1,013	6,311,172	NA	1,630 (2,804)	127 (903)	2:43:08	2:29:09
St. Matthew	372,767,445	7,016	8,432,423	0.7%	8,835 (14,343)	165 (910)	14:44:13	13:27:05

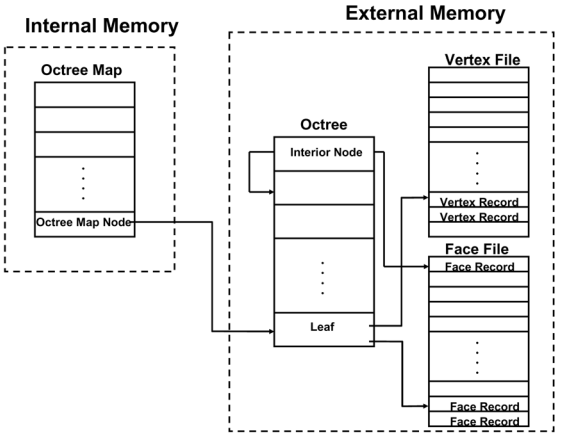

Fig. 1. Internal and external structures in the construction phase.

worse. Generating the mesh representation from a polygon soup requires a reindexing of the mesh, negating the efficiency gained from increased locality. Moreover, most meshes are distributed in an indexed format, so the generation of a polygon soup would require a time-consuming preprocessing phase.

The construction algorithm works in three phases: vertex scan, face scan, octree finalization. The vertex scan establishes occupancy in the octree and builds the vertex file. Each vertex is hashed into the bottom level of the octree map. Each entry in the map keeps a counter indicating the number of vertices it contains. An index is computed for each vertex consisting of the octindex of the cell into which it hashed plus the value of the cell's counter. We generate a record for each vertex consisting of this vertex index plus the coordinate data of the vertex. From this bottom octree map level, we propagate the vertex insertion upward. After the final vertex is processed, the octree map is scanned in a breadth first manner and disk space for the external octree is allocated along with disk space for the quadric file. At the end of this process, each node in the octree map corresponds to a node in the external octree and the index I (see Table 2) is resolved to the index of the external node. Within the external octree, for nonleaf nodes, the c_n and c_i fields are set to indicate the number and location, respectively, of the children of that node. The files are then memory mapped with read-write access.

The second phase of the construction process scans the mesh faces. This is time-intensive as it requires dereferencing the vertices of each face in order to compute an area-weighted normal vector and quadric matrix. Once these quantities are computed, we then hash each vertex of the face into the bottom level of the octree map and locate the external memory octree node associated with that vertex. The normal vector, the quadric, the vertex position, and a face count are accumulated in each octree node. A record is constructed for each face consisting of three octindices. We also determine the *split cell* in the octree which, when split into octants, causes the face to become nondegenerate. This operation can be done efficiently by unpacking the bits of the octindices for each vertex to generate three integer triples, with each triple corresponding to an octree node. If all of these triples are different, we right shift each integer in each triple and compare again. This shifting operation amounts to moving up a level in the octree. This process continues until at least two of the triples match, with the matching triples forming an octree index, and the number of shifts indicating a level in the octree. These values complete the face record. When all the faces have been written to disk, the file is sorted by split cell. This results in faces associated with the same split cell being grouped together and the cells being laid out in breadth-first order. A scan of the face file fills in the f_i and f_n fields for each octree cell. This allows a cell to locate the faces that are generated when it is split. The accumulated quadric, normal, face count, and vertex position data for each cell is propagated upward through the octree.

In the octree finalization phase, we compute the average normal and a proxy vertex position for each octree cell. The proxy vertex is found using the quadric matrix for each cell in the manner described by Lindstrom and Silva [18]. The quadric error for this vertex is then recorded in the field e. The next step is to compute a normal cone described by a half-angle a around the normal $\vec{n}$ for each octree node o. This is constructed as

$$a = \begin{cases} \max \angle\vec{n}, \vec{n_i} & \text{if o is a leaf} \\ \max(a_i + \angle\vec{n}, \vec{n_i}) & \text{otherwise,} \end{cases}$$

where, if o is a leaf, $\vec{n_i}$ are the normals of the faces it contains. If o is not a leaf, $\vec{n_i}$ and a_i are the normals cones of the children of o.

The construction process completes by externally sorting the vertex file using the vertex indices as keys. These keys are stripped out during the sort, leaving a file of coordinate triples sorted into sets corresponding to the cells they hash into in the bottom octree level. A scan of the vertex file sets the appropriate values in v_i and v_n, and quantizes the floating-point values.

3.3 Results

The theoretical running time for the construction algorithm is $O(V_{in} + F_{in})$, where V_{in} and F_{in} are the number of vertices and faces in the input mesh. This relies on the fact that the octree depth is at most 11 levels and assumes that the external sort is a radix sort. Our current implementation uses Linderman's rsort program [15], which, while actually a hybrid, still performs well. The memory requirements for the construction algorithm depend only upon the occupancy rate of the grid. If the grid contains C occupied cells, the algorithm will require $16C$ bytes of memory.

Some experimental results for the octree construction algorithm are in Table 3. The tests were run on a Linux box with a 1.8 GHz Athlon XP2500+ processor and 1GB of main memory. The timings show the process to be IO bound for all meshes larger than the Buddha, as IO time accounts for 75 percent or greater of the total time in each case. This is not entirely unexpected as the conversion to our external mesh format requires a significant amount of data be written to disk and sorted externally. The memory usage reported indicates the amount of storage used by the octree map, the only significant in-core data structure. Since data on disk are accessed as memory mapped files, the operating system uses any additional free memory to perform caching. This behavior can be seen in the peak memory usage values and effectively adapts the algorithm to its environment, allowing increased performance given additional free memory.

Among previous work, our external memory octree is most comparable to the XFastMesh [5] system in that it preserves access to the original surface data and produces a multiresolution data structure suitable for view-dependent rendering. The XFastMesh construction phase is performed in core, requiring a machine with a significant amount of virtual memory. In contrast, the amount of memory required to build our external memory octree is insensitive to the size of the input model. This enables our algorithm to process meshes with hundreds of millions of polygons on machines with a relatively modest amount of memory. As reported by DeCoro and Pajarola [5], the largest mesh they converted to the XFM format was the David 2mm model. The conversion took 49 minutes on a 450 MHz Sun Ultra60. Our algorithm processes the same model in 8 minutes, with the difference in speed attributable to the efficient use of memory and reliance on vertex clustering as opposed to edge contraction. While our experiment was run on a faster machine, our construction algorithm is IO bound and benefits little from the faster processor. The XFM file format is more concise than our external octree, requiring 241 MB as opposed to 435 MB to store the David 2mm mesh. However, as the size of the input model increases, the cost of storing the octree diminishes significantly as compared to the cost of storing the faces and vertices. For such models, the vertex quantization and face encoding we employ become much more effective in constraining the final disk usage. Moreover, while minimizing the disk footprint is important for large meshes, the current availability of cheap disk space makes it a less critical consideration than in the past.

Lindstrom [17] created a system with similarities to the one we propose. This work, developed concurrently with ours, is a completely out-of-core VDR system which employs an external memory octree. The most significant difference is that Lindstrom builds the octree from a simplified surface, resulting in the original surface no longer being accessible. The trade off is that the construction phase requires less time. When the size of the original and simplified meshes is close, the two systems perform similarly. Lindstrom reports it took 32 seconds to process the Buddha on a dual processor 800 MHz PIII system with 880 MB of memory. Our algorithm required 19 seconds on a more powerful uniprocessor system.

4 View-Dependent Rendering

One of the most important applications for the external memory octree is a view-dependent rendering system. Even with the incredible advances in graphics hardware, massive meshes have polygon counts high enough to make the overhead involved in view-dependent rendering worth the price.

4.1 Algorithm Overview

The mechanics of our VDR algorithm are similar to the one presented by Luebke and Erikson [19]. At each frame, we maintain a cut through the octree, with nodes along the cut forming a *front*. All their ancestor nodes are considered to be *active*. We maintain a front list and active list containing pointers to these nodes. The nodes on the active list contain the faces that are rendered in a given frame, while the nodes on the front form the vertex set of the mesh that is generated. Octree node data is read from disk using memory mapping.

TABLE 4
Data Stored in a VDR In-Core Octree Node

float[**3**]	v	proxy vertex position
float[**3**]	$\vec{n}$	normal vector
float	na	normal cone half-angle
short	v	reference count
short	l	level in octree
integer	I	index of external octree node
integer	o	octindex
boolean	$onactive$	flag for cell is active
boolean	$onfront$	flag for cell on the front
boolean	$contracted$	flag for contracted cell
boolean	$tested$	flag for cell view tested cell
pointer	$parent$	pointer to the parent node
list	$children$	list of pointers to children
octindex[**3**][]	$faces$	faces, as octindex triplets
offset[**3**][]	$offsets$	vertex offsets in vertex file
pointer[**3**][]	$resolvedfaces$	faces, as pointer triplets

The view-dependent rendering system memory maps the external memory octree file, the vertex file, and the face file. The latter two files can exceed the file size limit on some systems, requiring that the system split the files and memory map one piece at a time. The data kept in an in-core octree node is shown in Table 4. Exclusive of space used to hold data about faces, the size of a node is 60 bytes.

The active list is initially empty, while the front list is initialized to contain the octree root. At each frame, we use tests involving the current viewing parameters, described in Section 4.2, to determine whether to expand or collapse nodes along the front. If a node on the front passes the view tests, it is moved to the active list and its children are brought into memory and added to the beginning of the front list. If the node being expanded is a leaf, the children are simply vertices pulled from the vertex file. These are added to the list of children in the leaf node, but not added to the front list. The front will move down at most one level per frame. While this can result in some loss of quality for a given frame, the degradation generally only occurs when the user is moving the model rapidly. When movement slows so the model can be inspected, the relative shallowness of the octree allows the proper level of detail to be reached quickly.

If a node on the front fails the view test, the parent is then tested. If the parent fails, it is contracted; it is removed from the active list and added to the front. The process proceeds recursively on any children of that parent also on the active list. Children of the parent on the front are flagged for removal by setting the **contracted** bit. The flag bit **tested** is used to prevent multiple testing of parent nodes. We maintain a hash table, keyed by octindex and octree level, of pointers to nodes on the front. Any changes to the front require the table to be updated. In order to limit excessive waits due to paging, updating stops after 0.75 seconds elapse. Updating is interruptible and will continue from the stopping point when processing begins for the next frame.

After updating the front, the active list is scanned and the faces are rendered in immediate mode. Before rendering can take place, the vertices of each face must be resolved from octindex format to an octree node on the front. For nodes just added to the active list, we search the front hash table to find a pointer to a node on the front. The search

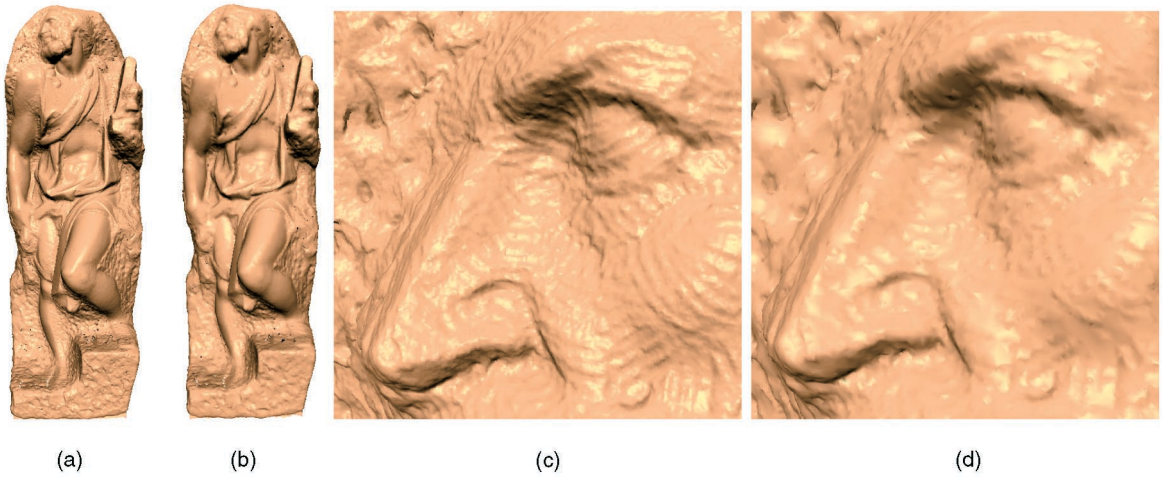

Fig. 2. The view-dependent renderer varies surface complexity based on approximate screen space error, lower T implies lower error. (a) $T = 1.0$, 400K faces. (b) $T = 2.0$, 150K faces. (c) $T = 0.5$, 300K faces. (d) $T = 2.0$, 100 K faces.

uses the octindex of the vertex and varies the octree level from deepest down to zero until a node is found. Since the octree has at most 11 levels, the search requires only a few constant time queries. If the vertex is resolved to an interior octree node, the proxy vertex of that node is used for rendering. If it maps to a leaf node that is active, the offset of the vertex is used to select the vertex data from the leaf node. For nodes that are not new to the active list, the vertices are already resolved to pointers. However, each pointer must be examined to ensure it still points to a node on the front. If it does not, meaning the front has moved, the search procedure outlined above occurs. Rendering the faces can be performed using the averaged normals from the octree per-vertex for smooth shading or by calculating per-face normals on the fly for flat shading.

4.2 View Dependence Tests

The view dependence tests applied to the octree nodes include view frustum culling, a backfacing test, and a screen space coverage test. The backfacing test uses the normal cone of the node, and is similar to that described by Luebke and Erikson [19]. One difference is that, instead of computing a bounding sphere for the geometry in the octree cell, we use a sphere located at the cell center with radius equal to the average distance from the cell center to the cell walls. A viewing cone is computed from the eyepoint to this sphere, with ϕ denoting the half-angle of the viewing cone. If θ is the angle between the cell normal and the view vector and n_a is the half-angle of the normal cone, a cell is considered backfacing if $\theta - \phi - n_a < \pi/2$. While the sphere will not closely approximate the geometry in the cell, it will generally be conservative; the approximating sphere will usually be at least as large as a tight sphere on the geometry. In tests using tight spheres, we saw no significant difference in visual quality. The effects of the backfacing tests on the David 2mm model are shown in Fig. 3. Frustum culling is accomplished by testing the sphere approximating an octree cell for intersection with the view frustum. An example of the results of this test can be seen in Fig. 4.

For the final view test, we compute the approximate screen space coverage of a node and compare it to a user-specified threshold T. Nodes with screen coverage less than T fail the test. Fig. 2 shows the effects of varying the screen space error threshold on the St. Matthew mesh. The approximating sphere computed in the backfacing test is used in this test as well. A well-known approximation [1] for the projected screen area of a sphere is $p = \frac{nr}{\mathbf{d}\cdot(\mathbf{c}-\mathbf{v})}$, where p is the radius of the projected sphere, $\mathbf{v}$ is the eye point, $\mathbf{c}$ is the sphere center, $\mathbf{d}$ is the normalized view direction, n is the distance from the viewer to the near plane, and r is the radius of the sphere. The screen area will then be πp^2 in a normalized device space (i.e., before the viewport transformation). Once again, the approximation error will be conservative as the sphere generally will be larger than a tight bounding sphere. To test the performance, we implemented a tight bounding sphere hierarchy. We also implemented an octree in which the cells were cubes, rather than parallelepipeds. The approximate sphere projection scheme had visual quality close to that of the cube-based octree and performed better than the point set bounding spheres, which coarsened too aggressively.

Used alone, the screen space coverage test essentially coarsens the mesh based only on distance from the eyepoint. We augment the test with a curvature heuristic we have developed which increases the error threshold for planar portions of the mesh and tightens the threshold in areas of high curvature. This is accomplished by comparing the projected screen area to $T_c = \frac{T\pi}{2n_a}$, where n_a is the normal cone half-angle for the octree cell being tested. The values of n_a are in the range $[0, \pi]$, with the larger values occurring in areas of high curvature. An example of the effect of the heuristic can be seen in Fig. 7. The heuristic clearly does a better job of allocating triangles to areas of high detail. The striations seen in the mesh generated without the heuristic are grid artifacts and are alleviated to a great degree by use of the heuristic.

4.3 Results

Table 5 shows performance statistics gathered during manually controlled viewing of the models on a system with a 1.8GHz Athlon processor, 1GB of memory, and a 128 MB GeForce FX 5200 display adapter. The numbers are averages from five trials for each model, each trial having the same frame count. The values listed in the View Time column include the time taken to perform the view-dependent tests and update the front, while the values

(a) (b)

Fig. 3. Surface unseen by the viewer is coarsened. (a) Visible surface. (b) Backfacing, unseen.

Fig. 4. Surface outside the view frustum is coarsened.

listed under Render Time denote the time taken to perform immediate mode rendering of the polygons. While the overhead is significant, the performance is still much faster than rendering the entire original mesh. The view-dependence tests are the current bottleneck in the system, with the IO time being reasonable when averaged across all the frames. However, paging can induce a performance hit when the view shifts significantly. Fig. 5 and Fig. 6 show frame time and IO time for viewings of the David 2mm model and the Lucy model. Significant activity occurs when the model is first loaded and thereafter when the user zooms in for a closeup look. Overall, the IO behavior is reasonable, with no thrashing occurring. This indicates that the layout of the octree on disk and use of memory mapped paging provides acceptable IO performance.

The performance of the VDR system can be compared to that of XFastMesh [5]. For the David 2mm model, XFastMesh rendered an average of 17,000 polygons per frame at approximately 27 frames per second. The overhead of their system was roughly 53 percent, running on a 450 MHz Sun Ultra60 workstation. Our VDR system rendered 78,059 polygons at 13 frames per second with an overhead of 60 percent. The faster graphics card on our system accounts for at least some of the higher polygon throughput of 1 million polygons per second versus 459,000. It is also likely responsible for the view tests requiring a higher percentage of the frame time in our system. XFastMesh uses a higher quality simplification algorithm than we do (edge collapse instead of vertex clustering), so it should be able to achieve higher quality images with lower polygon counts. At 8 ms per frame, the average IO time for our system was not significantly worse than the average block loading time for XFastMesh. This indicates that, at least in the case of the David 2mm mesh, using memory mapping to perform paging can offer performance close to that of a custom paging scheme. In any case, since the paging mechanism is orthogonal to our VDR algorithm, a custom paging scheme could be employed without any changes to the algorithm.

Comparison of the performance of our system with that described in [17] is problematic due to the differences in platform. On a dual processor 800 MHz Pentium III system with a GeForce3 graphics card and 880 MB of RAM, Lindstrom's VDR system has a peak throughput of three million triangles per second from a base mesh of around 2.7 million polygons. While the peak performance of our system on the David 2mm mesh is only around half of this, the difference is partly attributable to the end-to-end system running multithreaded on a two processor system. This allows view tests and rendering to be performed in parallel. Another consideration is that our base mesh for the David 2mm model is the original 8.2 million polygons, resulting in a larger octree. This results in more time spent paging in data and testing octree cells, but has the benefit of allowing greater visual detail.

5 Collision Detection

The second application we have developed for the external memory octree is a simple collision detection system for massive meshes. We perform approximate collision detection between models by using the octree cells as axis-aligned bounding boxes of vertex sets. The collision

TABLE 5
View Dependent Renderer Performance Running in an 800×600 Window

Mesh	Triangles	T	Nodes	Frame Time	View Time	Render Time	IO Time	Peak Memory (MB)
David 2mm	78,059	1.0	36,744	73 ms (13 fps)	43 ms (60%)	30 ms	8 ms	281
Lucy	85,698	1.0	43,017	89 ms (11 fps)	58 ms (65%)	31 ms	15 ms	371
St. Matthew	120,003	2.5	63,0651	250 ms (4 fps)	187 ms (75%)	73 ms	75 ms	590

Date are averages per frame, except for peak memory. IO time is included in view time.

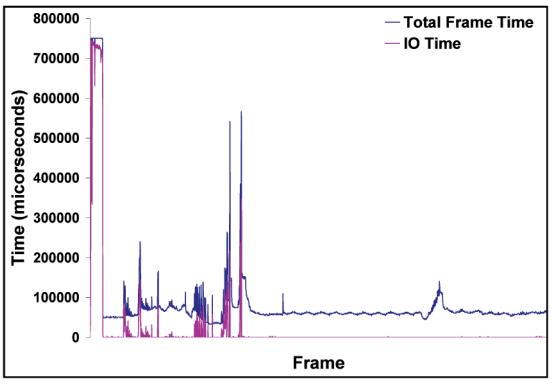

Fig. 5. Frame and IO time for viewing the David 2mm model.

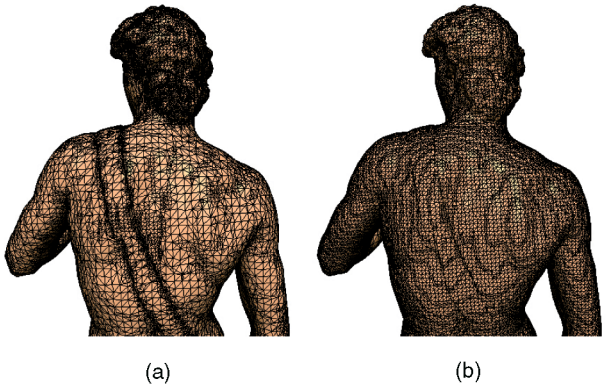

Fig. 7. The curvature heuristic creates an adaptive tessellation. (a) With heuristic, 50K faces. (b) No heuristic, 50K faces.

detection system recursively drills down, testing for cell intersections between octrees. In a typical collision detection system, intersection between two leaves would result in testing the polygons contained in each leaf against each other for intersections. Polygons in our octree are associated with their split cell, rather than a cell that completely contains them. As a result, polygon versus polygon intersection testing is problematic. Instead, our system simply reports a collision if two leaf nodes overlap.

When examining the error inherent in this algorithm, one should consider that the target meshes for this system are densely tessellated. For the Lucy mesh, only 1 percent of the triangles have edges long enough to penetrate more than two grid cells at the finest level of the octree. The David 1mm and St. Matthew meshes have no such triangles. If we assume no mesh edge spans more than two cells at the lowest level, we can characterize the possible error. False positive error occurs when two almost empty cells touch, yielding a maximum error of two times the length of a cell diagonal. For false negative error, the vertices of intersecting polygons must fall into nonintersecting cells. This implies that features smaller than two times the length of a cell diagonal can interpenetrate without being detected. Using cells in a $1,024^3$ grid carved from the mesh bounding box yields an error that will be a very small relative to the total mesh size. In our experiments, the error has generally not been noticeable. Our collision detection system focuses on object placement rather than physical accuracy. The system simply disallows mesh movements that would penetrate another object, placing the meshes back in the last recorded stable position. Some experimental results are shown in Fig. 8, which charts the total frame time and the time used for collision tests between two David meshes. These two models, together containing 16 million polygons, were rotated and translated in such a way as to periodically collide with each over 1,000 frames. While the overhead is noticeable, interactive frame rates are maintained throughout, and IO activity is still manageable. Fig. 9 shows a glancing blow between the two meshes, with the octree cells tested for a possible collision highlighted in red.

6 Simplification

The external memory octree data structure offers the ability to quickly generate single-resolution meshes that approximate the original input mesh to a specified accuracy. This simplification operation is similar to the work performed by the view-dependent refinement system. In both instances, a cut is formed through the octree, with cells along the front specifying vertex positions and cells behind the front specifying the faces. However, instead of using view related tests, the simplification operation uses geometric error in the form of the quadric error metric.

In our implementation, the user specifies an error threshold or a target number of faces. The front list initially contains just the root node. When using an error threshold, a breadth-first traversal of the octree expands nodes on the front with error exceeding the threshold. This continues until all front nodes exhibit error lower than the threshold.

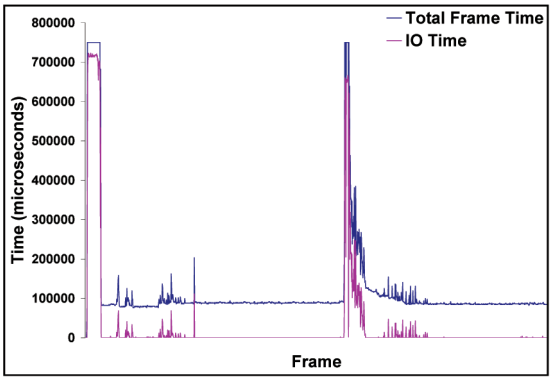

Fig. 6. Frame and IO time for viewing the Lucy model.

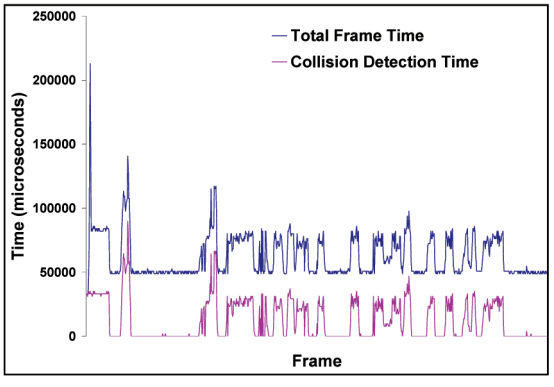

Fig. 8. David/David test, frame and collision detection time.

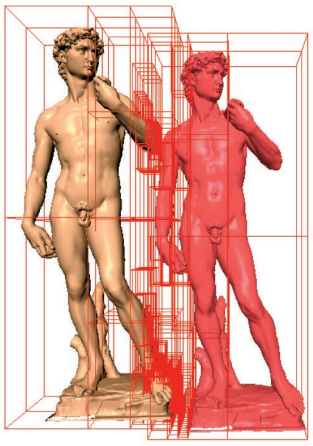

Fig. 9. Collision detection between David meshes.

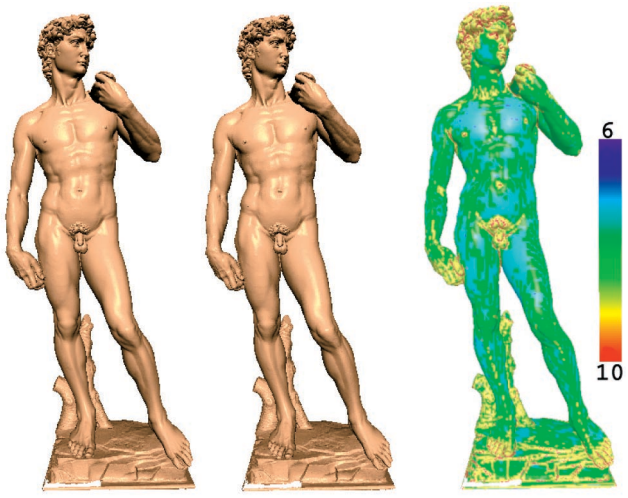

Fig. 10. Simplifying the David 2mm mesh. (a) shows the original mesh. (a) 8M faces. (b) 825K faces. (c) Octree depth.

Simplification using a target number of faces is similar, with the front list taking the form of a priority queue ordered by error. In this case, processing completes when the number of faces in the active list exceeds the target.

To produce the simplified mesh, the list of active cells is traversed and the octindices forming the faces are resolved to the appropriate proxy vertex along the front. The tool operates in one of two output modes. It can generate a polygon soup, writing out each face to disk as it as processed. An alternative mode produces an indexed face mesh. While this representation is more compact, generating it requires more memory as a hash table of previously encountered vertices must be kept during the resolution phase.

The simplification algorithm consumes $O(V_{out})$ memory, where V_{out} is the number of vertices in the approximation. The expected time bound when generating a specified number of faces is $O(V_{out}\log(V_{out}))$ due to the use of the priority queue. This assumes good hashing performance when resolving the vertex identities. An approximation produced by the simplification tool can be seen in Fig. 10, with some quantitative results shown in Table 6. The timings were taken on a 1.8 GHz Athlon XP2500+ processor using 1GB of memory and show our method to be faster than the uniform and multiphase clustering algorithms in [8]. The quality of the simplified mesh surpasses that produced by the uniform method, as the cut through the octree forms an adaptive approximation with bounded error. The 825,003 face mesh produced by our simplification algorithm exhibits about 68 percent of the average error in a similarly sized model produced by uniform clustering. Table 6 shows the average and standard deviation of depth in the octree for vertices in the approximation. The majority of vertices are at most three levels away from the average depth, indicating reasonable adaptivity. The processing times are also competitive with those reported in [13], where streaming simplification is applied to a mesh formatted as a processing sequence. On an 800 MHz Pentium III machine with 880 MB of RAM, their system simplified the David 2mm model to 825,415 faces in 3 minutes and 50 seconds (not including the preprocessing). Our tool generated a similarly sized simplification in less than one quarter of the time on a processor roughly twice as fast. While our simplification tool was faster, the streaming simplification algorithm employs edge contraction operations and will generate better quality approximations.

7 Conclusion and Future Work

We have described a new multiresolution data structure for representing massive polygonal meshes. Our proposed surface representation preserves access to the original surface data and allows efficient access to a rich set of sampled surface data. During the construction of this representation, memory usage is insensitive to the size of the input due to the use of a uniform grid to sample the surface. By building the representation in a memory efficient manner, the construction algorithm is capable of processing meshes containing hundreds of millions of polygons on commodity PC hardware. Using this data structure, we have developed algorithms for view-depen-

TABLE 6
Simplification Algorithm Performance

Mesh	Faces In	Faces Out	Time (h:m:s)	IO Time (h:m:s)	Memory (MB)	Avg Depth(σ)
David 2mm	8,254,150	500,000	47	42 (89%)	225	9.2 (3.3)
David 2mm	8,254,150	825,003	50	43 (86%)	233	9.5 (3.3)
David 1mm	56,230,343	1,000,000	11:22	11:04 (97%)	248	9.6 (2.9)
David 1mm	56,230,343	2,000,000	22:50	22:07 (97%)	340	9.9 (0.9)

dent rendering, approximate collision detection, and surface simplification. Empirical results show that these tools operate efficiently on massive meshes, demonstrating the flexibility and scalability of the multiresolution structure.

There remain several avenues for improvement on the structure we have presented. First, there remain opportunities for more aggressive compression of the representation. Also, the view-dependent rendering system could be improved by introducing multithreaded prefetching of cells from disk. We may also consider visibility-based culling of cells as this could allow the system to operate on meshes that fill space more densely.

ACKNOWLEDGMENTS

The authors would like to thank Marc Levoy and the people working on the Digital Michelangelo Project for making their models available to them. Also, they would like to thank Claudio Silva and Peter Lindstrom for making Linderman's rsort program available to them.

REFERENCES

[1] T. Akenine-Moller and E. Haines, *Real-Time Rendering.* A.K. Peters, Ltd., 2002.
[2] R. Borgo, P. Cignoni, and R. Scopigno, "An Easy-to-Use Visualization System for Huge Cultural Heritage Meshes," *Proc. 2001 Conf. Virtual Reality, Archeology, and Cultural Heritage,* pp. 121-130, 2001.
[3] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Adaptive TetraPuzzles—Efficient Out-of-Core Construction and Visualization of Gigantic Polygonal Models," *Proc. SIGGRAPH 2004,* Aug. 2004.
[4] P. Cignoni, C. Rocchini, C. Montani, and R. Scopigno, "External Memory Management and Simplification of Huge Meshes," *IEEE Trans. Visualization and Computer Graphics,* vol. 9, no. 4, pp. 525-537, Oct.-Dec. 2003.
[5] C. DeCoro and R. Pajarola, "XFastMesh: Fast View-Dependent Meshing from External Memory," *Proc. IEEE Visualization 2002,* pp. 363-370, 2002.
[6] J. El-Sana and Y.J. Chiang, "External Memory View Dependent Simplification," *Computer Graphics Forum,* vol. 19, no. 3, pp. 83-94, Aug. 2000.
[7] M. Garland and P.S. Heckbert, "Surface Simplification Using Quadric Error Metrics," *Proc. SIGGRAPH 1997,* pp. 209-216, Aug. 1997.
[8] M. Garland and E. Shaffer, "A Multiphase Approach to Efficient Surface Simplification," *Proc. IEEE Visualization 2002,* pp. 117-124, 2002.
[9] M. Guthe, P. Borodin, and R. Klein, "Efficient View-Dependent Out-of-Core Visualization," *Proc. Fourth Int'l Conf. Virtual Reality and its Application in Industry (VRAI 2003),* Oct. 2003.
[10] H. Hoppe, "Progressive Meshes," *Proc. SIGGRAPH 1996,* pp. 99-108, Aug. 1996.
[11] H. Hoppe, "View-Dependent Refinement of Progressive Meshes," *Proc. SIGGRAPH 1997,* pp. 189-198, Aug. 1997.
[12] H. Hoppe, "Smooth View-Dependent Level-of-Detail Control and Its Application to Terrain Rendering," *Proc. IEEE Visualization 1998,* pp. 35-42, 516, Oct. 1998.
[13] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink, "Large Mesh Simplification Using Processing Sequences," *Proc. IEEE Visualization 2003,* 2003.
[14] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk, "The Digital Michelangelo Project: 3D Scanning of Large Statues," *Proc. 27th Ann. Conf. Computer Graphics and Interactive Techniques,* pp. 131-144, 2000.
[15] J Linderman, "rsort Man Page," June 2000.
[16] P. Lindstrom, "Out-of-Core Simplification of Large Polygonal Models," *Proc. SIGGRAPH 2000,* pp. 259-262, July 2000.
[17] P. Lindstrom, "Out-of-Core Construction and Visualization of Multiresolution Surfaces," *Proc. 2003 Symp. Interactive 3D Graphics,* pp. 93-102, 2003.
[18] P. Lindstrom and C.T. Silva, "A Memory Insensitive Technique for Large Model Simplification," *Proc. IEEE Visualization 2001,* pp. 121-126, 2001.
[19] D. Luebke and C. Erikson, "View-Dependent Simplification of Arbitrary Polygonal Environments," *Proc. SIGGRAPH 1997,* pp. 199-208, Aug. 1997.
[20] R. Pajarola, "Fastmesh: Efficient View-Dependent Meshing," *Proc. Pacific Graphics,* pp. 22-30, 2001.
[21] C. Prince, "Progressive Meshes for Large Models of Arbitrary Topology," Master's thesis, Univ. of Washington, 2000.
[22] E. Shaffer and M. Garland, "Efficient Adaptive Simplification of Massive Meshes," *Proc. IEEE Visualization 2001,* pp. 127-134, 2001.
[23] A. Wilson, E. Larsen, D. Manocha, and M.C. Lin, "Partitioning and Handling Massive Models for Interactive Collision Detection," *Computer Graphics Forum (Eurographics '99),* P. Brunet and R. Scopigno, eds., vol. 18, no. 3, pp. 319-330, 1999.
[24] J. Wu and L. Kobbelt, "A Stream Algorithm for the Decimation of Massive Meshes," *Graphics Interface 2003 Proc.,* 2003.
[25] J.C. Xia and A. Varshney, "Dynamic View-Dependent Simplification for Polygonal Models," *Proc. IEEE Visualization 1996,* pp. 327-334, Oct. 1996.

Eric Shaffer is a PhD student in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He received the BS degree in mathematics and computer science from the University of Illinois at Urbana-Champaign and the MS degree in computer science from the University of Minnesota at Minneapolis. His research interests include out-of-core algorithms, scientific and information visualization, mesh processing, and parallel computation. He is a student member of the IEEE and the ACM.

Michael Garland received the BS degree in mathematics and computer science from Carnegie Mellon University in 1993 and the PhD degree in computer science in 1999. He is currently an assistant professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests include mesh processing, real-time rendering, 3D object modeling, scientific visualization, and out-of-core algorithms.