In Proceedings IEEE Visualization 2002, pages 363–370. IEEE Computer Society Press, 2002.

XFastMesh: Fast View-dependent Meshing from External Memory

Christopher DeCoro Renato Pajarola

Computer Graphics Lab
Information & Computer Science
University of California, Irvine
cdecoro@uci.edu, pajarola@acm.org

ABSTRACT

We present a novel disk-based multiresolution triangle mesh data structure that supports paging and view-dependent rendering of very large meshes at interactive frame rates from external memory. Our approach, called XFastMesh, is based on a view-dependent mesh simplification framework that represents half-edge collapse operations in a binary hierarchy known as a merge-tree forest. The proposed technique partitions the merge-tree forest into so-called detail blocks, which consist of binary subtrees, that are stored on disk. We present an efficient external memory data structure and file format that stores all detail information of the multiresolution triangulation method using significantly less storage then previously reported approaches. Furthermore, we present a paging algorithm that provides efficient loading and interactive rendering of large meshes from external memory at varying and view-dependent level-of-detail. The presented approach is highly efficient both in terms of space cost and paging performance.

CR Categories: I.3.3 [Computer Graphics]: Image Generation—Display Algorithms I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Surface and Object Representations E.5 [Files]: Organization/Structure— [H.3.2]: Information Storage and Retrieval—Information Storage

Keywords: level-of-detail, multiresolution modeling, out-of-core rendering, interactive large-scale visualization

1 INTRODUCTION

With the rapid advances in 3D scanning technology, it has become possible to generate geometric models much larger than the capacity of physical main memory (see [12]). Therefore, rendering these models with standard level-of-detail (LOD) approaches and in-memory multiresolution frameworks causes uncontrolled paging of the operating system's virtual memory manager that negatively affects the interactive display. In order to use traditional polygonal methods to render such large objects and surfaces at interactive frame rates, we require:

1. an efficient view-dependent multiresolution triangulation and rendering framework that can be dynamically modified, and
2. support for rendering from external memory which includes paging of view-dependent LOD mesh data from disk.

External-memory algorithms and data structures [1, 2] present a unique set of challenges not encountered in traditional main-memory techniques. Some efforts have been made in scientific visualization to develop external-memory algorithms (see abstract [18]). In real-time LOD rendering of large polygonal meshes from external memory the main challenges are related to maintaining and synchronizing two separate datasets: the in-memory triangle mesh data structures, and the on-disk data.

Figure 1: A model of Michaelangelo's David statue, comprising over 8.25 million polygons. The model is shown as simplified with XFastMesh. The full-quality (within view frustum) smooth-shaded model is shown on the left, and the wire-framed, flat-shaded image on the right is shown to display the simplification. The view frustum is focused on the head and upper body.

In this paper we present *XFastMesh*, a system which allows efficient view-dependent rendering of very large triangle meshes from external memory, such as Michaelangelo's statue of David as shown in Figure 1, which consists of 8.25M polygons. The system's in-memory view-dependent rendering method is based on FastMesh [19], however, incorporates substantial changes to dynamically update the main memory mesh data structures. Additionally, the main contribution of XFastMesh is an external memory data structure (*XFM* file format) and paging algorithm that supports efficient loading of mesh data from disk at different LODs.

The remainder of the paper is organized as follows. In Section 2 we review closely related work and preliminary background. Section 3 describes the external memory data structures and file format, and Section 4 explains the in-memory data structures and rendering algorithm. Section 5 provides experimental results and Section 6 concludes the paper.

2 RELATED WORK

2.1 View-dependent Meshing

Several view-dependent mesh simplification and rendering methods have been proposed in the literature so far. The approaches presented in [6, 25, 26] and [9, 10] are based on a binary vertex hierarchy derived from iteratively simplifying the input mesh by edge collapse operations [8]. Nodes of this hierarchy are collapsed or refined for each rendered frame based on view-dependent error metrics such as projected edge length or approximation error. The triangle mesh is dynamically updated and rendered according to changes in the hierarchy.

Generalized view-dependent rendering frameworks based on arbitrary vertex hierarchies are presented in [16] and [22]. While not optimized for storage cost or rendering performance these generic approaches support a wide range of simplification operations. The method presented in [3] is based on simple vertex insertion and removal operations and provides a compact representation, however, at the expense of mesh update and rendering performance.

Specialized view-dependent terrain triangulations based on height-field models are presented in [4, 11, 14] and [10]. Besides [10], these approaches take advantage of the regular grid structure of elevation data.

2.2 Out-of-core Mesh Simplification

Only recently, research in multiresolution modeling has concentrated on out-of-core mesh simplification. The methods presented in [13, 15] and [23] provide efficient simplification of very large meshes on external memory using vertex clustering. However, these methods do not provide a multiresolution triangle mesh structure for efficient view-dependent rendering nor do they support paging from disk.

The approach presented in [5] presents the first method for out-of-core triangle mesh simplification and view-dependent rendering with different LODs from external memory. Additionally, this method provides an out-of-core preprocess that could also be used by our approach to generate an edge-collapse hierarchy. Compared to [5] our approach is significantly better in terms of storage cost and rendering performance as shown in the experimental results section. In [20] a progressive mesh approach is presented which is based on mapping a set of array ranges of the entire progressive mesh sequence into contiguous virtual memory. No rendering performance results are reported in [20] and loading detail information from disk reportedly causes rendering pauses.

2.3 FastMesh

The concepts of view-dependent meshing and rendering of XFastMesh are based off the FastMesh method [19]. FastMesh is a real-time view-dependent progressive meshing system based on a hierarchy of edge collapse operations similar to [26] and [9]. FastMesh is different from other approaches in its use of an efficient half-edge data structure to represent the triangle mesh and the simplification operations. This allows for compact representation and efficient mesh updates. An edge is represented by two reversely directed half-edges. Each triangle is represented by its three counter-clockwise oriented half-edges as shown in Figure 2. As shown in [24], a directed half-edge h contains a reference to its start vertex h_{vertex}, information on its previous h_{prev} and next h_{next} half-edges within the same triangle, and its reverse half-edge h_{rev} in the adjacent triangle.

With *merge tree forest* we refer to the binary half-edge collapse hierarchy of FastMesh [19]. As shown in Figure 3, a particular LOD triangulation is defined by an *active front* through the merge tree forest which represents all nodes that are subject to collapse or split operations. At run-time this front is traversed and updated according to view-dependent error metrics for each frame, and the corresponding triangle mesh is rendered. The view-dependent error metrics used in XFastMesh are largely identical to [19] and require two parameters to be stored with each node in the merge tree forest, the bounding sphere radius and bounding normal cone opening angle.

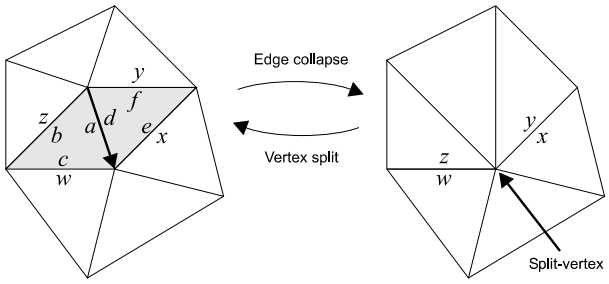

Figure 2: An edge collapse removes a vertex and two faces from the mesh, which are reintroduced by its dual vertex split. In this example, the half-edge A is collapsed, removing half-edges A through F. W, X, Y, Z are considered *adjacent* to the removed half-edges, and remain in the mesh after the collapse.

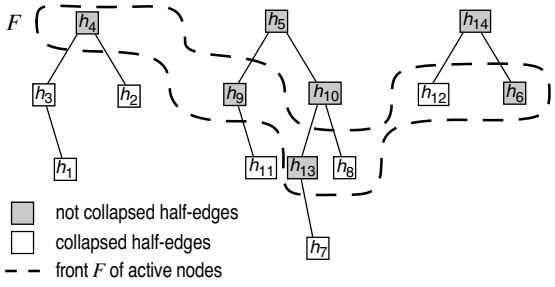

Figure 3: An example merge tree forest, with the current front represented by the dashed line. This active front represents the current level of detail, and can be moved up (lower detail) and down (higher detail) as the situation requires.

3 EXTERNAL-MEMORY STRUCTURES

The external-memory layout of XFastMesh is designed to minimize space usage while attempting to store all related information together on disk, thereby minimizing the amount of disk accesses required to render a scene. Especially given that most external memory devices, such as hard disk drives, are *block devices* and read information in blocks of data, we break the XFastMesh data structures into logically-grouped blocks to take advantage of this fact.

The XFastMesh file format (XFM) as shown in Figure 4 contains the following data fields:

1. Fixed-size file header, which describes the position and length of other data fields

2. Initial vertex coordinates and face information; those vertices and faces that are present in the coarsest mesh, and are not introduced by any vertex split operation.

3. Root list, which lists the blocks containing the root nodes of the merge tree forest.

4. Block index table, which provides the offset and length of each detail block in the file. Because we use a variable length block size, we cannot determine detail block positions in the file implicitly from the block numbers, and use the block index for this purpose.

5. Detail blocks, which comprise the majority of the file, and contain information for LODs, including vertex coordinates and edge split/collapse information.

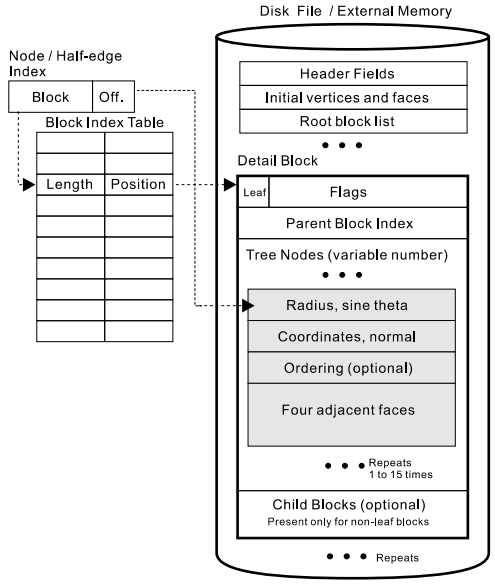

Figure 4: XFastMesh external memory structure. The block number and offset of a node are determined from its index. The block index table is used to determine the offset of a block within the file. Note that the Block Index is stored as part of the file, but is accessed as a memory-mapped file, rather than with explicit I/O commands.

3.1 Initial faces and vertices

Detail information such as vertex and normal coordinates are generally stored on disk with the merge tree node that will introduce the corresponding vertex into the mesh by means of a vertex split. Each vertex split inserts one vertex and two triangles into the mesh as shown in Figure 2. Because the faces and vertices in the coarsest mesh, referred to as *initial* faces and vertices, are not created by any vertex splits, they must be stored separately in the file. For these, the data file contains a list of vertex coordinates, their associated normal vectors, and triangle faces as specified in the file header. This information is used to generate the initial coarse mesh on system initialization. These initial faces and vertices will be kept in main memory during system run time.

3.2 Root block list

On system initialization, the algorithm will load the blocks corresponding to the root nodes of the merge tree forest, in order to set up the initial active front. To identify these nodes and their corresponding detail blocks efficiently, the file stores a list of all *root blocks*, those blocks that contain the root nodes of the merge tree forest. The system reads this list on startup and loads the specified blocks.

3.3 Block index

The primary purpose of the XFastMesh data file is to store a large number of detail blocks (see Section 3.4) which contain information for different LODs. Each detail block is identified by a numerical index, and the algorithm will use this index to locate blocks on disk. For space efficiency reasons, detail blocks do not have a fixed size and thus cannot be located on disk using implicit position information only. Instead we keep a fixed table, referred to as the block index I, that lists the on-disk location and the length of each block, which are represented with two scalars per block. This block index table is accessed by every block load operation to find the block location on external memory.

Because this table is fixed in size, immutable, and usable directly in its on-disk storage format, we access the table by mapping the corresponding data of the file into virtual memory using the file memory-mapping capability (such as the `mmap` function under UNIX). Alternatively, since the block index is sufficiently small it can be loaded entirely into main memory without causing undue space overhead.

3.4 Detail blocks

The primary units in the XFastMesh data file are referred to as *detail blocks*. Each detail block contains the information required to describe a complete subtree of the merge-tree forest. Thus the binary merge-tree hierarchy is regularly partitioned into binary subtrees as shown in Figure 5.

Within each detail block, explicit links between nodes are minimized using an array representation of the complete binary subtree. Tree nodes are numbered and stored in a breath-first ordering such that given the index i of a particular node, the parent node can be found at position $(i-1)/2$, and the left and right child nodes at $i \cdot 2 + 1$ and $i \cdot 2 + 2$ respectively. This *detail-block tree* structure allows us to store $2^l - 1$ tree nodes per block, where l is number of levels in the binary subtree.

Each detail block in the tree contains the same number of nodes. The value of l is user specified; in the current implementation, $l = 4$ has shown efficient results. Lower values for l allow for finer granularity in memory allocation, at the expense of more frequent loading of small detail blocks.

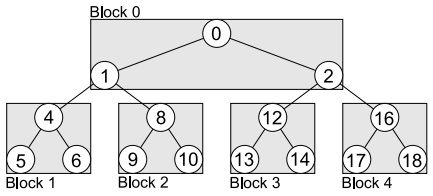

Figure 5: Detail block tree structure of a 4-level merge tree, with $l = 2$. The nodes in each block are numbered such that the block number is $b = n/2^l$ for a given node n. Block 0 contains the root node of its tree and thus is a root block. Its index stored in the file's root list, as described in Section 3.2

A given detail block b consisting of l levels of a binary tree, contains the following fields:

1. A set of flags to indicate if the block is a leaf block, and to indicate which nodes are present in this block.

2. Index of the parent block of b.

3. A variable-length array N of merge-tree nodes, with $|N| \geq 1$ and $|N| \leq 2^l - 1$.

4. An optional array C containing indices of child blocks (if not a leaf block).

XFastMesh can use the complete subtree structure of a block to implicitly represent *intra-block links* between nodes within the same detail block. However, the parent-child relationships between blocks, called *inter-block links*, must be explicitly stored in each block. Each detail block stores the index of its parent node, as well as the indices of each of its 2^{l+1} child nodes.

In a simple implementation of detail blocks that consist of complete binary trees, significant storage space is wasted for leaf blocks that are not full (not all available nodes are used). To resolve this internal fragmentation, the nodes of each block are fully packed as shown in Figure 6. Each block contains a 8- 16- or 32-bit flag (for $l = 3, 4, 5$, respectively) that indicates which nodes are present in this block. The loading algorithm uses this information to properly reconstruct the incomplete binary subtree in main memory.

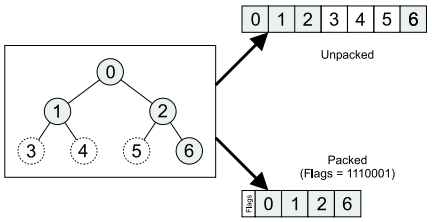

Figure 6: A partially filled detail block (unused nodes displayed in dashed outline) can be packed in order to use less space, using bitflags to identify which nodes are present.

Additional efficiency issues are related to the fact that a block tree has a large fan-out (a block with $l = 4$ has 8 nodes in the lowest level, and thus $2^l = 16$ child blocks). For shallow and unbalanced merge-tree hierarchies this results in a large ratio of leaf to non-leaf blocks. The space used to store the array of child pointers C is wasted in each leaf block since there are no child blocks to reference. Thus with a large percentage of leaf blocks, storing child pointers in every one of these would result in a significant amount of storage overhead. We avoid storing these child pointers by using the high-order bit of a block's flag-set to indicate whether or not that block is a leaf. Thus we store the array C of child block indices only for non-leaf blocks.

The most significant data field in a detail block is the array N of merge-tree nodes. For each particular node $n \in N$ and its corresponding half-edge h, we store the following information:

1. Vertex coordinates for $h.vertex$ (12 bytes)
2. Normal vector coordinates for $h.vertex$ (12 bytes)
3. Bounding-sphere radius and sine of bounding normal cone angle (as described in [19]) (8 bytes)
4. The indices w, x, y, z (from Figure 2) of the four half-edges adjacent to the two triangles introduced by expanding h (16 bytes)
5. (Optional) Index of n in the global ordering of edge collapses (used for fold-over prevention, 4 bytes)

Each node in a detail block consumes 52 bytes. Potentially, this can be reduced to under 36 bytes by quantizing the normal, bounding cone, and bounding radius as in QSplat [21], and removing the global ordering.

3.5 Data file construction

The XFastMesh data file is generated in a preprocess from a given set of half-edge collapse operations that define a binary merge-tree forest. The details of generating such a progressive mesh simplification are independent of XFastMesh which focuses on creating the out-of-core data structures. A progressive mesh simplification using edge collapses can be constructed in various ways (for references see [7, 17]). The procedure for creating the data file is as follows:

1. Create the merge tree hierarchy from the input data
2. Collapse mesh into the lowest resolution configuration
3. Write all present vertices and faces into the file as initial vertices and initial faces
4. Traverse the tree in depth-first order and renumber the half-edges corresponding to each node such that for a node numbered n, its corresponding block $b = n/(2^l - 1)$ (further described in Section 4).
5. Again traverse the tree in depth-first order, writing each block out to the data file, and storing its block-index table entry.
6. Write the block-index table to the file
7. Determine the block numbers of each root block; write the root list into the file

A key point is that the algorithm must traverse the tree twice: before writing the blocks to disk, it must renumber the nodes in the blocks so as to achieve an implicit relationship from a node to its corresponding block. Once the nodes have been renumbered, they can be written to the file.

4 MAIN-MEMORY STRUCTURES

Previous mesh simplification systems assume that the dataset will fit entirely in memory, allowing all data structures to be allocated up-front and stored in arrays. This has the benefit of allowing constant time indexing and access to edges or tree nodes.

In order to avoid allocating all the required memory at once, but maintaining the ability to perform constant time indexing, we use a page-table-like structure to organize data as shown in Figure 7. This structure, known as the *detail block directory* D stores pointers to all loaded detail blocks. Furthermore, this indirect indexing structure allows us to easily add and delete blocks at run-time.

Constant-time indexing of nodes and half-edges is achieved by using a consistent numbering for the blocks and half-edges, as if each block were full. Therefore, for $2^l - 1$ nodes per block, the global number of the first node in this block modulo $2^l - 1$ is equal to 0, as was shown in Figures 5 and 6. Each vertex split operation creates two triangle faces, and so each node corresponds to six half-edges. Therefore, with $m = 6 \cdot (2^l - 1)$ of half-edges per block, the number of the first half-edge in this block modulo m is equal to 0. Given a half-edge number h, we can compute its block number $b = h/m$, and its offset as h modulo m. This numbering is consistent for all blocks, even for blocks that are not full.

During runtime, XFastMesh will need to access a particular node based only on its index n. To find this node, we need to compute the block number $n_{block} = n/(2^l - 1)$ of this node, and the offset $n_{offset} = n$ modulo $(2^l - 1)$ of the node within the block. The node n can then be accessed as $D[n_{block}][n_{offset}]$.

For faster implementation, we can number each block as if the number of tree nodes it contains is an even power of two. This allows us to use shifts and bitmasks to substitute the more expensive division and modulo operations. Essentially, we assign a certain number of bits to identify the block, and the remaining bits to identify the offset within the block. This technique is much faster than the hashing used in other external memory methods [5].

The choice of l, the number of levels per block, affects the size of the directory. Smaller values of l will result in more blocks, and require more entries in D; conversely, larger values of l will result in less blocks and a smaller memory footprint for D. Because the directory is always stored in main memory, a memory-limited system might choose a large value of l to reduce this fixed memory overhead.

Once a detail block has been loaded from disk, it is inflated into its full main-memory representation. We copy most of the basic fields from the on-disk representation. Additionally, we create a set of half-edges to represent the faces, with each half-edge containing a pointer to its corresponding start vertex, and the number of its reverse half-edge. In the on-disk representation we only store the four reverse half-edges w, x, y, z adjacent to the two faces of a node, as shown in Figure 2. In the main-memory representation, we use this information to re-create all six half-edges. While the disk representation uses the implicit complete tree representation to store node connectivity (intra-block links), we explicitly store parent, left and right pointers for each node in main memory, as well as left and right child block numbers for efficiency.

Figure 7 illustrates the structure of detail blocks as used in main memory. For each in-memory tree node n, and its associated half-edge h, XFastMesh contains the following information and uses a total of 108 bytes per node:

1. Pointers to the parent, left and right children of n in the merge tree (12 bytes)
2. Vertex coordinates for $h.vertex$ (12 bytes)
3. Normal vector coordinates for $h.vertex$ (12 bytes)
4. Bounding-sphere radius and sine of bounding normal cone angle (8 bytes)
5. Global index of h (4 bytes)
6. Left (*lblock*) and right (*rblock*) child block indices (8 bytes)
7. Six half-edges (including h) representing two faces; each half-edge contains a vertex pointer and the index of its reverse (8 bytes each; 48 bytes total)
8. (Optional) Index of n in the global ordering of collapses (used for fold-over prevention, 4 bytes)

As each node corresponds to a vertex, there are roughly as many nodes as there are vertices in the input mesh. Therefore, we can conclude that XFastMesh uses 108 bytes per vertex in main memory, plus some small overhead for the fixed directory structure. We use 4 bytes per block in the block index directory, to store a pointer to the particular detail block. We also store 4 bytes per block for a timestamp, which is used for block caching and removal as discussed in Section 4.2. As in the out-of-core data structure discussed in Section 3.4, we can encode certain values more efficiently through quantization if necessary.

4.1 Block loading

For interactive rendering, the system requests blocks from disk in two different situations at run-time:

1. When the front of active nodes moves down the tree, requiring more detail.
2. When a split operation on a node causes a *forced split* of other nodes.

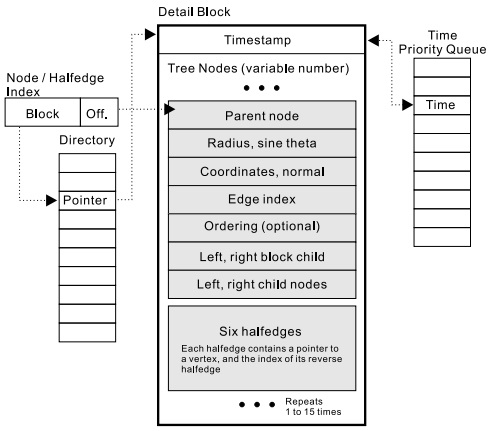

Figure 7: Main memory structure of XFastMesh, for $l = 4$. Once block number and offset are determined from a node index, XFastMesh uses the Directory to find the individual detail block. Each directory entry may point to an individual detail block. Each detail block will have a corresponding entry in the time priority queue used for caching and paging.

For each frame, the algorithm will test each node in the active front, to determine if it should be split or collapsed. If it has to be split, the traversal will recursively descend the tree to test child nodes. As it descends, it will reach the *frontier* of the loaded blocks, the lowest blocks currently loaded into the tree, and will trigger the first case. The data structure indicates these frontier nodes by setting the $n.left$ and $n.right$ pointers of the node n to *null*, while setting the $n.lblock$ and $n.rblock$ values to the actual child block index. Once the algorithm detects this case, it will load both the left and right child blocks into memory, and continue.

Alternatively, a node split operation may cause a forced split, in which a node from another part of the merge-tree forest must be split first, and its information loaded from disk. A forced split is detected by the absence of any face referenced by the reverse half-edges of a, b, c, d in Figure 2, and it causes the insertion of this face into the triangle mesh by the corresponding node split. In this case, the algorithm will use the numerical index of the face to determine which block to load, and recursively load the ancestors of this block until that subtree can be connected to the existing hierarchy in main memory. This requires that for a given block b, we must determine the index of the node n that is the parent of b, such that $n.left = b.root$ or $n.right = b.root$. We store this index of a block in $b.parent$, and we use the directory structure to access the node, and link the new block into the existing hierarchy. If the directory entry for the block containing n, $D[n_{block}]$, is *null*, then we must recursively load this block and its ancestors.

4.2 Block deletion

In order to avoid loading a block from disk every time it is used, XFastMesh caches blocks using a least-recently-used (LRU) strategy. Each currently loaded block is given a timestamp that indicates the last frame in which it was part of the active front as shown in Figure 7.

XFastMesh makes use of two quotas that determine its memory usage. The *soft* quota, q_s is a suggestion to the system to remove the least recently used blocks once the total number of loaded blocks b_{loaded} has exceeded q_s. XFastMesh will begin removing blocks as soon as $b_{loaded} > q_s$, but if the quota does not allow for enough blocks to achieve the specified view-dependent LOD threshold, it will load additional blocks as required and exceed the quota.

In contrast, the *hard* quota, q_h is a set limit on the number of loaded blocks. Once XFastMesh has reached its hard quota, it will

```
function mesh-update()
   for each node in active-front
      test-node(node)

function test-node(node)
   split = view-test(node)
   if( split )
      if( node has unloaded child blocks )
         load blocks into memory
      if( node is currently collapsed )
         split-node(node)

      test-node(node->left)
      test-node(node->right)
   else
      collapse-node(node)
   end if
end function
```

Figure 8: XFastMesh update algorithm. We analyze each node in the active front, and test to see if the front should be moved up or down. We assume that `view-test` is a function that takes a merge-tree node as input, and returns as output whether or not that node should be split. `split-node` and `collapse-node` are functions that perform their respective split or collapse operations on their input node.

not load additional blocks until more storage has been made available.

This design allows for flexibility in memory allocation. To force the system to always remove unused blocks, set $q_s = 0$. To keep all blocks until a set limit is reached, set $q_s = q_h$. To always load blocks as needed, the hard quota can be disabled by setting $q_h = \infty$. Note that q_s cannot be set higher than q_h.

Candidates for block removal are only detail blocks whose nodes are all below the active front (see also Figure 3). For these blocks, their root nodes are collapsed and the siblings of those nodes are also collapsed. The node m is a sibling of node n if $m.parent = n.parent$. At run-time, when a node is collapsed, the system checks to see if this condition is met. If so, its block is timestamped with the current time, based on the number of frames rendered, and placed in a minimum priority queue as shown in Figure 7. When nodes are split, their corresponding blocks and their siblings are removed from the priority queue since they are no longer candidates for removal.

4.3 Mesh update

The mesh update operation consists of two steps, shown in Figure 8. First, the algorithm traverses the active front, performing a test on each node to determine if that node should be split or collapsed (see [19] for details on that test operation). The algorithm will descend the tree recursively as needed. Secondly, once the algorithm has determined that a merge tree node should be modified, it will perform the split or collapse operation on that node. As the algorithm descends the tree, it will eventually reach the frontier of the loaded blocks. At this point, the algorithm will determine if the frontier nodes have additional child blocks, and if so, it will load those blocks into memory, connect them into the tree, and continue recursively testing.

4.4 Rendering

A given XFastMesh configuration has two types of triangle faces that it must render:

1. Initial faces, which are part of the coarsest mesh and never removed, are loaded at system start.

```
function mesh-render()
   for each initial face
      render initial face as a triangle
   for each root-node in merge-forest
      render-node(root-node)
end function

function render-node(node)
   if( node is split )
      render both faces
      render-node(node->left)
      render-node(node->right)
   end if
end function
```

Figure 9: XFastMesh rendering algorithm. The initial faces are rendered first, followed by the detail faces, which are rendered by traversing the merge tree forest

2. Detail faces, which are added by vertex splits to the mesh, are loaded as part of detail blocks.

All faces are stored as a sets of half-edges, with a face defined as a set of three consecutive half-edges. The half-edges of the initial faces are stored separately from those of the detail faces, in a fixed array. The rendering algorithm proceeds through this array, rendering each of the initial faces from their constituent half-edges.

In contrast, the detail faces are stored in the dynamic merge tree structure, with each node in the merge tree containing six half-edges, corresponding to two faces. Detail faces are rendered by recursively traversing the merge tree forest top-down, and drawing the two triangles that correspond to each node that is split. The rendering algorithm is shown in Figure 9

5 EXPERIMENTAL RESULTS

All benchmark testing was performed on a Sun Microsystems Ultra60 workstation, with the Solaris 8 operating system, running at 450MHz. The amount of RAM usable to the application is set by the hard quota q_h.

5.1 Storage Requirements

As shown in Table 1, the XFastMesh data files are very efficient; the Stanford Bunny model represented in XFM format uses 2.04 MB, versus 3.03 MB for the original (ASCII) PLY file. Note that the indexed face representation of the PLY format does not include any multiresolution triangle mesh or view-dependent rendering information whatsoever. XFastMesh is approximately 350% more efficient than the implementation of external-memory rendering framework proposed in [5] (VDT) which uses 7.1 MB for the same model. With only about 60 bytes per vertex, XFM is also more efficient than the file based FastMesh format (FM) [19], the view-dependent progressive meshes of [9], or the multitriangulation approach [3].

Much of this storage efficiency can be credited to packing incomplete blocks as described in Section 3.4. For the horse model, with 97K faces, XFastMesh generates 12231 blocks, with $l = 4$. Each block could potentially store $2^l - 1 = 15$ nodes. If the blocks were completely filled, we would have about 6600 blocks. It follows that the 12231 blocks are only about half full, therefore, the packed storage scheme saves approximately 50% the storage cost of the unpacked scheme. Removing unnecessary child pointers also proved to save a significant amount of disk storage. From experimental results, the horse model, which uses 2.84 MB without unnecessary child pointers, previously used over 3.5 MB, an increase of about 25%. For that particular model, testing showed that out of 12231 total blocks 10358 were leaf blocks (or about 85%).

Model	Faces	PLY	FM	XFM	B/Δ	VDT	B/Δ
bunny	69K	3.0	2.1	2.0	29.6	7.1	102.9
knee	75K	3.2	2.3	2.2	29.6	3.4	45.3
horse	100K	4.1	2.9	2.8	28.4	-	-
ball joint	274K	12.4	8.2	8.0	29.3	27.4	100.0
dragon	202K	7.3	-	-	-	21.8	107.9
dragon	871K	33.8	26.2	25.6	29.4	-	-
buddha	293K	-	-	-	-	31.7	108.2
buddha	1087K	42.6	32.6	31.8	29.3	-	-
david	8254K	343.0	247.6	241.4	29.2	-	-

Table 1: File sizes in megabytes for various models. The table displays the number of faces in each model, and the size of file in megabytes for source ASCII format (PLY), FastMesh format (FM), XFastMesh format (XFM) and in the external-memory format as presented in [5] (VDT).

5.2 Runtime Costs

Before rendering a mesh, XFastMesh first must preprocess the input mesh in order to build the merge-tree forest similar to [19] (FM), and then create the XFM file. Creating the XFM file adds only minor overhead to the preprocess, as seen in Table 2.

Rather than use a greedy approach to selecting edge collapses, XFastMesh iteratively selects the largest independent sets of edge collapses. Because of that XFastMesh does not need to constantly re-sort the list of edge collapses, as other techniques do using a heap structure, which would result in an $O(n\ lg\ n)$ running time for the preprocess. Instead, because XFastMesh only needs to sort the list some constant number of time, the number of iterations, and all of the keys, the approximation error values, are scalar numbers, XFastMesh can use a linear-time radix sort. Therefore, the running time of the preprocess is $O(n)$, where n is the number of triangles in the input mesh. Note that the one exception to the linear time growth is the david model, in which case the system is forced to page in memory.

Model	Faces	FM	μs/Δ	XFM	μs/Δ
bunny	69K	8.4	122	8.6	125
knee	75K	8.8	117	9.4	125
horse	97K	12.0	120	12.3	123
ball joint	274K	39.6	145	40.2	146
dragon	871K	119	137	122	140
happy	1087K	149	137	154	142
david	8254K	2628	318	2994	363

Table 2: Preprocessing times in seconds for various models. The basic view-dependent simplification preprocess is shown as FM, the preprocess plus the external file creation step is shown as XFM

At rendering time, because of the use of external memory, XFastMesh is able to load into main memory only the data needed at any given time for a particular viewpoint and error tolerance. This also results in extremely quick start-up times for viewing an initial coarse representation of the displayed mesh. Even for the largest of our test meshes, the David statue, our application only requires a couple of seconds to load the coarsest mesh, initialize the merge tree forest and display the first frame.

To determine the runtime performance of XFastMesh, we rendered each of our models, with the viewpoint moving on a fixed path. The screen-error tolerance was set to 0.02. This tolerance controls the view-dependent mesh simplification according to the screen-space error metric as described in [19]. The soft quota was set to $q_s = 10\%$.

In Table 3, we display the per frame timing in milliseconds for each of the basic tasks performed by XFastMesh, as exhibited on our test models. We recorded the timing using the high-resolution timer functionality provided by the Solaris operating system. These tasks include:

- View tests, performed on each node to determine if it should be split or collapsed (column VT)
- Mesh updates, which reconfigure the mesh through either edge collapses or vertex splits (column U)
- Block loading, loading a block from disk and converting it to its main-memory representation (column BL)
- Rendering of the selected triangle mesh (R)

The Table 3 also displays the average number of triangles shown per frame (#Shown) and the average number of updates performed per frame (#U). The results show that XFastMesh is very efficient, and scales well to large meshes.

Model	Faces	#Shown	#U	U	VT	BL	R
bunny	69K	10K	1593	4ms	3.5ms	11ms	9ms
knee	75K	10K	589	4ms	3.2ms	8ms	9ms
horse	97K	9K	270	15ms	3.0ms	7ms	15ms
ball joint	274K	14K	463	6ms	5.3ms	5ms	13ms
dragon	871K	39K	1700	11ms	9.1ms	12ms	25ms
happy	1087K	38K	1110	19ms	15ms	5ms	35ms
david	8254K	17K	1148	10ms	7.5ms	2ms	17ms

Table 3: Runtime performance measures for the basic operations in XFastMesh. The numbers shown are the average number of faces and updates per frame, and the time cost per frame of updates, view-tests, block loading and rendering in milliseconds.

Figure 10 provides several examples of a large model rendered at interactive frame rates with XFastMesh. The error tolerance was configured to adjust automatically to match a specific target frame rate; in this case, the target was set to 5 frames/second to achieve high-quality images while maintaining interactive rendering. The figure demonstrates the high-detail view from the user's perspective, as well as showing the entire mesh to demonstrate the effectiveness of the simplification.

6 CONCLUSION

In this paper we presented efficient algorithms and data structures for out-of-core multiresolution meshing and view-dependent rendering. Our approach called XFastMesh provides a very efficient representation of the multiresolution triangle mesh on external storage, requiring only about 60 bytes per vertex. This out-of-core data structure allows for efficient view-dependent retrieval and paging of triangle mesh data and supports interactive rendering of large-scale meshes. Our in-memory multiresolution mesh data structure allows for paging mesh update information from disk. This changes dynamically at run-time, and supports fast view-dependent mesh refinement and rendering. Experiments on a variety of triangle meshes have shown the efficiency of our approach in terms of space cost (size of data structure) and rendering performance (including paging from disk).

XFastMesh currently does not include an out-of-core preprocess to generate the XFM format but relies on doing that off-line for a particular data set on a machine with sufficient virtual memory. With the target of the XFM file format, future work will address the issue of efficient out-of-core generation of the multiresolution mesh simplification hierarchy.

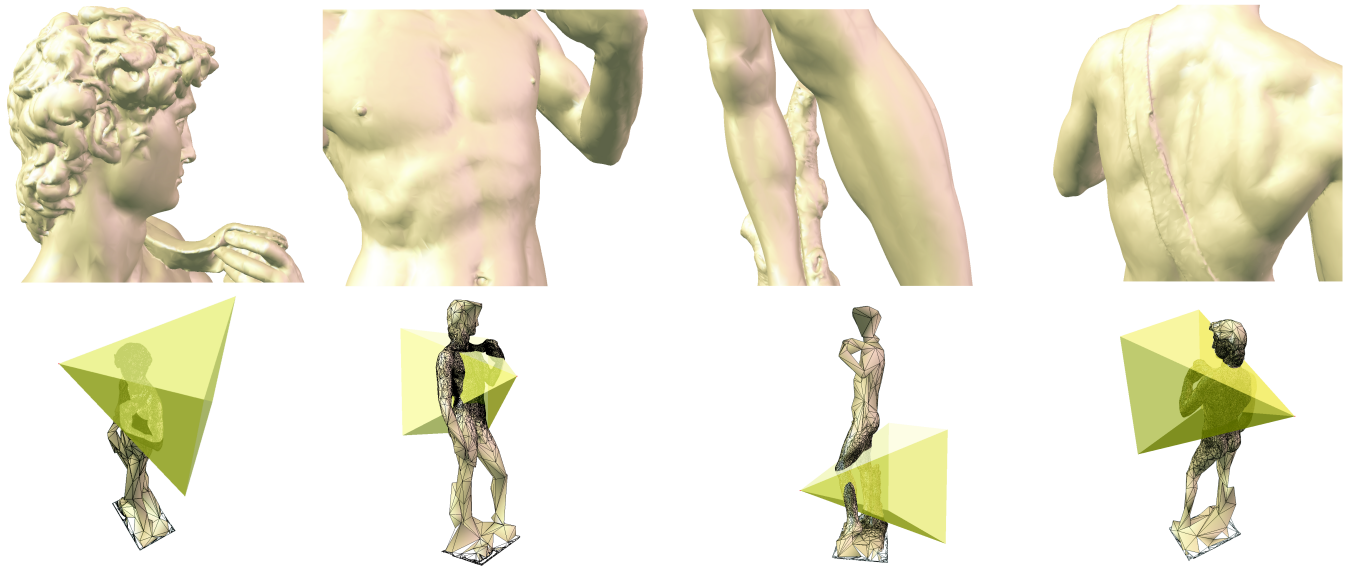

Figure 10: Four renderings of Michaelangelo's David statue, taken at high-quality settings with the error tolerance set to achieve rendering times of approximately 5 frames/second. The top row shows the view from the user's perspective; the bottom row shows the same scene outside the user's view frustum, which is represented by a yellow pyramid. From left to right, the error tolerance τ and number of triangles Δ are: (head) $\tau = .00728$ $\Delta = 67092$, (chest) $\tau = .00260$ $\Delta = 69428$, (legs) $\tau = .00559$ $\Delta = 50828$, (back) $\tau = .00337$ $\Delta = 67356$

ACKNOWLEDGMENTS

We would like to thank the UC Irvine UROP program for providing funding for this research. We would also like to thank the Stanford Computer Graphics Lab[1], the Georgia Tech Large Geometric Models Archive[2] and Cyberware[3] for providing freely-available high-resolution geometric models.

REFERENCES

[1] J. Abello and J. S. Vitter. *External Memory Algorithms*. American Mathematical Society, Providence, R.I., 1999.

[2] L. Arge. *Efficient External-Memory Data Structures and Applications*. PhD thesis, Department of Computer Science, University of Aarhus (Denmark), 1996.

[3] L. De Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulations. In *Proceedings IEEE Visualization 98*, pages 43–50, 1998.

[4] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. Roaming terrain: Real-time optimally adapting meshes. In *Proceedings IEEE Visualization 97*, pages 81–88, 1997.

[5] J. El-Sana and Y.-J. Chiang. External memory view-dependent simplification. In *Proceedings EUROGRAPHICS 2000*, pages 139–150, 2000.

[6] J. El-Sana and A. Varshney. Generalized view-dependent simplification. In *Proceedings EUROGRAPHICS 99*, pages 83–94, 1999.

[7] P. S. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. SIGGRAPH 97 Course Notes 25, 1997.

[8] H. Hoppe. Progressive meshes. In *Proceedings SIGGRAPH 96*, pages 99–108. ACM SIGGRAPH, 1996.

[9] H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings SIGGRAPH 97*, pages 189–198. ACM SIGGRAPH, 1997.

[10] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings IEEE Visualization 98*, pages 35–42. Computer Society Press, 1998.

[11] R. Klein, D. Cohen-Or, and T. Huttner. Incremental view-dependent multiresolution triangulation of terrain. In *Proceedings Pacific Graphics 97*, pages 127–136. IEEE, Computer Society Press, 1997.

[1] http://www-graphics.stanford.edu/data/3Dscanrep/

[2] http://cc.gatech.edu/projects/large_models/

[3] http://www.cyberware.com

[12] 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. In *Proceedings SIGGRAPH 2000*, pages 131–144. ACM SIGGRAPH, 2000.

[13] P. Lindstrom. Out-of-core simplification of large polygonal models. In *Proceedings SIGGRAPH 2000*, pages 259–262. ACM SIGGRAPH, 2000.

[14] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. In *Proceedings SIGGRAPH 96*, pages 109–118. ACM SIGGRAPH, 1996.

[15] P. Lindstrom and C. T. Silva. A memory insensitive technique for large model simplification. In *Proceedings IEEE Visualization 2001*, pages 121–126. Computer Society Press, 2001.

[16] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings SIGGRAPH 97*, pages 199–208. ACM SIGGRAPH, 1997.

[17] D. P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics & Applications*, 21(3):24–35, May/June 2001.

[18] K.-L. Ma. Large-scale data visualization. *IEEE Computer Graphics & Applications*, 21(4):22–23, July-August 2001.

[19] R. Pajarola. FastMesh: Efficient view-dependent meshing. In *Proceedings Pacific Graphics 2001*, pages 22–30. IEEE, Computer Society Press, 2001.

[20] C. Prince. Progressive meshes for large models of arbitrary topology. M.S. Thesis, 2000.

[21] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings SIGGRAPH 2000*, pages 343–352. ACM SIGGRAPH, 2000.

[22] D. Schmalstieg and G. Schaufler. Smooth levels of detail. In *Proceedings VRAIS 97*, pages 12–19, 1997.

[23] E. Shaffer and M. Garland. Efficient adaptive simplification of massive meshes. In *Proceedings IEEE Visualization 2001*, pages 127–134. Computer Society Press, 2001.

[24] K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics & Applications*, 5(1):21–40, January 1985.

[25] J. C. 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*, 3(2):171–183, April-June 1997.

[26] J. C. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *Proceedings IEEE Visualization 96*, pages 327–334. Computer Society Press, 1996.