Eurographics/ IEEE-VGTC Symposium on Visualization 2010
G. Melançon, T. Munzner, and D. Weiskopf
(Guest Editors)

Volume 29 (2010), Number 3

Out-of-Core Simplification and Crack-Free LOD Volume Rendering for Irregular Grids

Zhiyan Du[†] and Yi-Jen Chiang[‡]

Polytechnic Institute of New York University, Brooklyn, NY, USA

Abstract

We propose a novel out-of-core simplification and level-of-detail (LOD) volume rendering algorithm for large irregular grids represented as tetrahedral meshes. One important feature of our algorithm is that it creates a space decomposition as required by I/O-efficient simplification and volume rendering, and simplifies both *the* internal *and* boundary *portions of the sub-volumes progressively by edge collapses using the (extended) quadric error metric, while ensuring any selected LOD mesh to be* crack-free *(i.e., any neighboring sub-volumes in the LOD have consistent boundaries, and all the cells in the LOD do not have negative volumes), with all computations performed I/O-efficiently. This has been an elusive goal for out-of-core progressive meshes and LOD visualization, and our novel solution achieves this goal with a* theoretical guarantee *to be crack-free for* tetrahedral *meshes.*
As for selecting a desirable LOD mesh for volume rendering, our technique supports selective refinement *LODs (where different places can have different error bounds), in addition to the basic* uniform *LODs (where the error bound is the same in all places). The proposed* scalar-value range *and* view-dependent selection *queries for selective refinement are especially effective in producing images of the highest quality with a much faster rendering speed. The experiments demonstrate the efficacy of our new technique.*

1 Introduction

The rapid growth of the data size in recent years has posed a big challenge to scientific visualization. In this paper, we intend to attack this challenge by proposing a novel *out-of-core* simplification and level-of-detail (LOD) volume rendering algorithm. We focus on the class of *irregular grids* represented as tetrahedral meshes, which is the most general class of volumetric data and arises in applications such as computational fluid dynamics, shock physics, and so on.

In order to perform LOD rendering out-of-core, it is necessary to store the multiresolution representation in blocks. To facilitate I/O-efficient simplification and volume rendering, these blocks need to correspond to the sub-volumes obtained by some space partition of the volume. Commonly used such structures include octrees, *kd*-trees, and so on. However, the neighboring sub-volumes in the desired LOD may lie at different levels in the tree, causing the *boundary consistency* problem. Technically, it is desirable to avoid this problem and achieve a *crack-free* LOD mesh, namely, any neighboring sub-volumes in a selected LOD have *consistent boundaries*, and all the cells in the LOD are *fold-over free* (i.e., do not have negative volumes). A simple solution (as in [SS06b]) would be not to simplify the boundary cells until at a higher level where these cells become interior to an ancestor sub-volume. But for those cells that are initially cut at the top level, they cannot be simplified until at the tree root (i.e., at the very end of simplification), which is undesirable. It turns out that this issue of crack-free LOD in the out-of-core setting for *general* meshes is technically quite challenging; previously there were only a few techniques addressing this issue ([CGG*04, YSGM04, CGG*05, SM05] for *triangle* meshes and [SS06a] for *tetrahedral* meshes). However, it is not guaranteed that the boundary cells can always be simplified; even when they can, typically they have to be simplified in *future levels* rather than at *every current level*; and sub-volumes of disparate errors could be merged and simplified, resulting in less-smooth simplifications (see Sec-

[†] zdu@cis.poly.edu; research supported by NSF grant CCF-0541255.
[‡] yjc@poly.edu; research supported in part by NSF CAREER Grant CCF-0093373 and NSF Grant CCF-0541255.

tion 2). Therefore out-of-core crack-free LOD has been an elusive goal, especially for *tetrahedral* meshes that are usually *highly irregular*.

Our algorithm has an important feature that it achieves this goal mentioned above. Specifically, it creates a space decomposition as required by I/O-efficient simplification and volume rendering, and guarantees that *both* the *internal* and *boundary* portions of the sub-volumes can be simplified *at every current level*, with any selected LOD mesh guaranteed to be crack-free. Moreover, we use *error-based* merging to merge and simplify sub-volumes of similar errors for higher quality, where our simplification is progressive by edge collapses using the (extended) quadric error metric [GZ05]. With these properties, our technique tries to fill in the gap in the literature and complement the previous approaches.

As for selecting a desirable LOD mesh for volume rendering, our technique supports *selective refinement* [CFM*04] that allows an LOD to have *varying* error bounds over different places, in addition to the basic *uniform* LOD (the same error bound over all places). We give a flexible scheme for selective refinement queries. In particular, the proposed *scalar-value range* and *view-dependent selection* queries can be guided by a quick low-LOD image (to find the colors of important features together with the color mapping in the transfer function, and/or to find the desirable viewing parameters), and are especially effective in producing images of the highest quality with a much faster rendering speed.

Our LOD-mesh approach is independent of the final volume rendering method, and hence any tetrahedral volume rendering engine can be used as the volume renderer. In this paper we use the HAVS code [CICS05], which is a state-of-the-art tetrahedral-mesh volume rendering technique using programmable GPU. We modify it to make it run out-of-core, which we call *out-of-core HAVS*.

The experiments demonstrate the efficacy of our new technique. In particular, for full-resolution volume rendering on datasets much larger than main memory, we can improve the running time over *out-of-core* HAVS (without LOD) from 19.71 minutes to 3.82 minutes using selective-refinement LOD with almost the same image quality.

2 Previous Work

Recently a huge amount of work has focused on irregular-grid volume rendering using programmable GPU. As a comprehensive review of them is not our focus, we refer to the excellent survey [SCCB05] and the review in [MHDH07]. As mentioned, we use the *HAVS* method [CICS05], which we review in Section 3.3.

While GPU-based techniques can achieve interactive rendering for moderate-size datasets, LOD or multiresolution approaches [LRC*02] are important for large datasets. However, there are relatively few results in this area for tetrahedral meshes. A related topic, tetrahedral-mesh simplification, was developed in [THJW98, SG98, THJ99, CM02]. Other techniques with topology considerations were given in [CL03] and the references therein. These algorithms do not support LOD volume visualization. In [CFM*04], a multiresolution representation for tetrahedral meshes is built to support selective refinement queries for visualization. The work [CCSS05] introduced a *sample-based* simplification for LOD volume rendering; the simplification is done by sampling the mesh triangle faces with the connectivity removed.

The techniques mentioned so far are all in-core algorithms. For out-of-core approaches, early results [FS01, CFSW01] gave out-of-core volume rendering techniques for tetrahedral meshes without supporting LOD or using GPU. Recently, a *streaming* technique was given in [VCL*07] to simplify tetrahedral meshes, without producing a multiresolution representation or supporting LOD rendering. A streaming compression method was given in [ILGS06]. Various clustering approaches for out-of-core tetrahedral meshes was discussed in [DDPS05]. A *progressive* volume rendering algorithm [CBPS06] was proposed for the client-server model, where the client has limited main memory but the preprocessing and the major run-time computations need to be performed *in-core* on the server; only a single resolution is kept for the data. The *iRun* approach [VCS*07] is an out-of-core extension of the *sample-based* LOD method [CCSS05], where no connectivity or mesh LOD is maintained. As mentioned in Section 1, [SS06b] performs out-of-core simplification and LOD volume rendering for tetrahedral meshes, but it uses the simple approach of *not* simplifying the sub-volume boundaries at all. The only such approach that addresses the issue of crack-free LOD is [SS06a], which is based on [CGG*05] and we discuss them together below.

There has been an extensive work on out-of-core, LOD view-dependent rendering on polygonal models; we refer to [YL06, CRMS03] and their references. As mentioned in Section 1, among the space-decomposition methods only those in [CGG*04, YSGM04, CGG*05, SM05] address the issue of crack-free LOD for general 3D triangle meshes. In Quick VDR [YSGM04], the boundary between neighboring nodes are allowed to be simplified and a dependency between these two nodes is created if there is no other "rippling" new dependency created. However, there is no systematic way on how to stop the rippling. In Progressive Buffers [SM05], clusters are simplified one by one in an arbitrary order with all neighboring clusters entirely loaded to main memory. The current cluster can simplify its boundary together with the sharing neighbor, whose interior stays fixed. However, two or more clusters can impact the boundary of the same neighbor at different times, making the simplification non-trivial—LOD mesh must follow the same sequence of simplifications to obtain consistent boundaries. But this issue is not discussed. Also, for highly irregular volume meshes the cluster complexity can vary greatly; loading

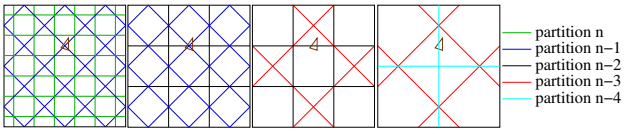

Figure 1: *An example of the scheme in [CGG*05, SS06a] where the triangle cannot be simplified at all levels shown.*

neighbors entirely could be inefficient and memory expensive.

The Batched Multi Triangulation [CGG*05] (whose basis is the in-core *directed acyclic graphs (DAG)* method [DM02]) improves upon TetraPuzzles [CGG*04], and the same idea is used in the Segment-Based Tetrahedral Meshing [SS06a]. Intuitively, the scheme uses a sequence of coarser (and rotating) space partitions, where at each level of simplification two consecutive partitions are *super-imposed* to each other to form the active partition (Fig. 1). Only the triangles/cells lying entirely *within* a region of the active partition can be simplified. With the restriction that the neighboring nodes in LOD have level difference *no larger than one*, the scheme achieves crack-free LOD. With an "ideal" partition sequence the cells crossing the region boundary can be simplified at a few levels up (2 levels at best). However, it cannot be guaranteed that boundary cells can always be simplified (see Fig. 1: the triangle stays for many levels). Moreover, two sub-meshes of disparate errors could be merged and simplified since the merging is based on a pre-decided space partition rather than on errors of sub-meshes, resulting in less-smooth simplifications. In contrast, our new technique resolves these issues along different directions.

3 Our Approach

3.1 Overview

LOD Cuts and Error-Based Merging Our key design principle is to exploit the structure of the *error-based LOD cuts* to achieve crack-free LODs. In the preprocessing phase, we build a tree, called *merge tree M*, by first partitioning the input mesh spatially into sub-volumes (corresponding to tree leaves) with roughly the same number of vertices, and then simplifying and merging these (leaf) sub-volumes *bottom-up* using an edge-collapse method. During the entire process of tree construction, the boundary among neighboring sub-volumes at each step is kept consistent. Because we simplify one tree node (sub-volume) at a time, we achieve boundary consistency by propagating the boundary edge collapses of the current sub-volume to its neighboring sub-volumes. Each time we create a new tree node by merging some sub-volumes and simplifying it, a new, corresponding boundary version (i.e., the boundary status resulting from the simplification) is also created. In this way, if we create n new nodes, counting the original version we have $n+1$ boundary versions. As we will see later, if we create the new nodes *in the order of increasing errors*, then for all possible query error bounds ε, there are only $n+1$ possible LOD cuts (a breadth cut on tree M) satisfying ε, corresponding to those $n+1$ boundary versions. On the other hand, in order to achieve smooth, continuous LOD meshes (it is important to reduce the cell/triangle-face count in volume rendering since we need visibility sorting), we also further simplify each sub-volume's interior separately and keep a *progressive* representation (see below). Therefore, to get both consistent boundary and continuous LODs, we have two phases of simplification for each tree node (sub-volume): (1) simplifying *globally* including the interior and boundary, in the order of increasing errors, to get a new boundary version and a simplification error lower bound ε_l; (2) simplifying the *interior* further in the same order, to get a simplification error upper bound ε_u. The resulting sub-volume is called the *base mesh* of this tree node. Each node is bounded in error range $[\varepsilon_l, \varepsilon_u]$, meaning that it can be *continuously* simplified/refined within this error range using progressive representation.

The tree M is just a *tree skeleton*, storing only the minimum amount of information needed in each node (e.g., $[\varepsilon_l, \varepsilon_u]$, the $[\min, \max]$ scalar values and the axis-aligned bounding box of its sub-volume); the actual sub-volume meshes are stored *separately* on disk. Therefore each node is quite small and we assume that the tree M can entirely fit in main memory. In fact we first define a suitable number of vertices in each initial sub-volume (e.g., 20K), which decides the number L of leaf sub-volumes in M and thus the size of M, so that M can entirely fit in main memory (M is at most 82.5KB in all our experiments).

In the run-time phase, we first decide the desired LOD cut on tree M satisfying query error ε. Once the cut is chosen, the boundary version is chosen too. As mentioned above, this boundary version is one of the $n+1$ versions constructed during preprocessing and is crack-free. Within each cut, we can have continuous versions of LODs by *independently* and progressively refining each node in the cut. This can be used to support *selective refinement* LODs.

Progressive Boundary and Interior with Fire Wall It is easy to see that a tree node (sub-volume) can have neighbors at many different tree levels, therefore it needs many boundary versions to be consistent with the neighbors. We achieve this by using a *progressive representation*, i.e., for each tree node we store the *base mesh*, and keep a linear sequence of edge collapses propagated from neighbors (which will be *in the order of increasing errors* in a nice way; see later), so that we can follow the sequence linearly to simplify the (boundary of the) base mesh. Also, as mentioned, to support continuous LODs we want to refine the interior of the nodes in the LOD cut continuously. Again we use a progressive representation, keeping a linear sequence of *vertex splits* (the inverse of edge collapses due to the *internal*

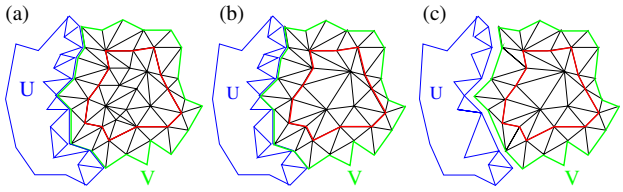

Figure 2: *The firewall. For sub-volume V, the outer-boundary is shown in green and the firewall is shown in red. (a)* After *the* global *simplification on V; the firewall is identified at this point. (b) After the* internal *simplification on V, i.e., the* base mesh *of V. (c) After the* global *simplification of neighbor sub-volume U. The boundary edge collapses of U are propagated to V to simplify the outer-boundary of V. Note that the firewall is* intact.

simplification in phase (2), in *totally sorted* order) to refine the interior.

This scheme requires the ability to simplify/refine the boundary and interior *independently*. The idea is to keep the border between them *intact* during simplification/refinement so that it acts as a *firewall* to protect both portions. In a sub-volume V, a vertex shared by other sub-volume(s) or a vertex on the boundary of the input mesh is called an *outer-boundary* vertex; the remaining vertices of V that are connected to the outer-boundary vertices are called the *firewall* vertices (Fig. 2). The edges shared between sub-volumes (i.e., connecting shared vertices) are the *boundary edges* that will need to be propagated to neighbors if collapsed. The cells connecting between the outer-boundary and the firewall form the *boundary portion* of V, and the remaining cells form the *internal portion/interior* of V.

We stress that we first perform a *global* simplification on V, which is *not restricted by the firewall* and can already simplify V as desired. Then we identify the firewall and perform the internal simplification. Therefore the firewall does not prevent us from performing a desired smooth simplification.

Referring to Fig. 2, where we first simplify V and then U. In (a), the global simplification on V has been done (which also propagated boundary edge collapses to U to make the boundary consistent). (When globally simplifying V, for an edge (a,b) where only a is on the outer-boundary, we only allow collapsing b to a so that it will *not* create a new vertex in any neighbor sub-volume.) We then identify the firewall and perform internal simplification on V (see (b)). During internal simplification, for a candidate edge e, if there is one endpoint on the firewall, then we collapse the other endpoint toward the firewall endpoint so that the firewall is intact; if both endpoints are on the firewall then we do not collapse e. In (c), after the *global* simplification of U, its (outer-)boundary is also simplified, and the boundary edge collapses are propagated to V to make the boundary consistent. Note that these boundary edges only affect the outer-boundary of V (see (c)) and again the firewall is intact.

We keep a *separate* boundary portion so that when we need the *boundary* information from neighbors we can avoid loading the whole neighbor and be I/O-efficient.

3.2 Preprocessing Phase: Out-of-Core Simplification

There are two major tasks in our preprocessing algorithm: (1) mesh partition, and (2) error-based sub-volume merging.

3.2.1 Task 1: Mesh Partition

Initially, we use the *meta-cell* technique [CSS98] to partition the input mesh into L sub-volumes of roughly the same size, each consisting of spatially neighboring cells. These sub-volumes will correspond to the L leaves of the merge tree M. Every cell is uniquely assigned to one single sub-volume; vertices shared by two or more sub-volumes are duplicated into each sub-volume. Such sub-volumes are defined as *neighbors*. We keep the global vertex ID for each shared vertex so that they can be identified from neighbors.

We also build a *connectivity graph* G, where the nodes of G are the sub-volumes and the edges of G connect neighbors. If two sub-volumes are merged during simplification we merge the corresponding nodes with their edge in G collapsed (see Fig. 3). In this way we maintain the neighboring information among the existing sub-volumes.

3.2.2 Task 2: Error-Based Sub-volume Merging

In this step, we repeatedly simplify and merge sub-volumes bottom-up to build the merge tree M. We perform half-edge collapses using the (extended) quadric error metric [GZ05].

Recall from Section 3.1 that for each node of M we have an error range $[\varepsilon_l, \varepsilon_u]$ where ε_l is the error after the global simplification and ε_u is the error after the additional internal simplification; clearly $\varepsilon_l \leq \varepsilon_u$. At the leaf level, we set $\varepsilon_l = 0$ and only perform the *internal* simplification (until no more than c vertices remaining for a parameter c (e.g., 1.5K)) and set up ε_u. To continue, we put all current nodes to a priority queue Q with ε_u as the key, and repeatedly extract minimum from Q. The first extracted node is put to a place holder P. In general, for the current extracted node w, we look at P to see if there are any neighbors of w. If not, we put w to P and repeat; otherwise we take out all neighbors of w from P and merge them all together with w to form a new tree node x, which becomes the parent of the merged nodes (e.g., in Fig. 3(b) node 2 is merged with node 1 from P to form node 6). Note that w has the largest ε_u value among its siblings; we set $\varepsilon_l(x) = \varepsilon_u(w)$, since some part ($w$) of the merged sub-volume already has errors up to this value (e.g., in Fig. 3(b) node 6 has $\varepsilon_l = 2$). We then apply the *global* simplification on the merged sub-volume up to error bound $\varepsilon_l(x)$ (which also propagates the collapsed boundary edges to the affected neighbors). Finally we fix the *firewall* and perform the *internal* simplification until no more than c vertices remain and set up $\varepsilon_u(x)$. At this point, the creating of new node x is com-

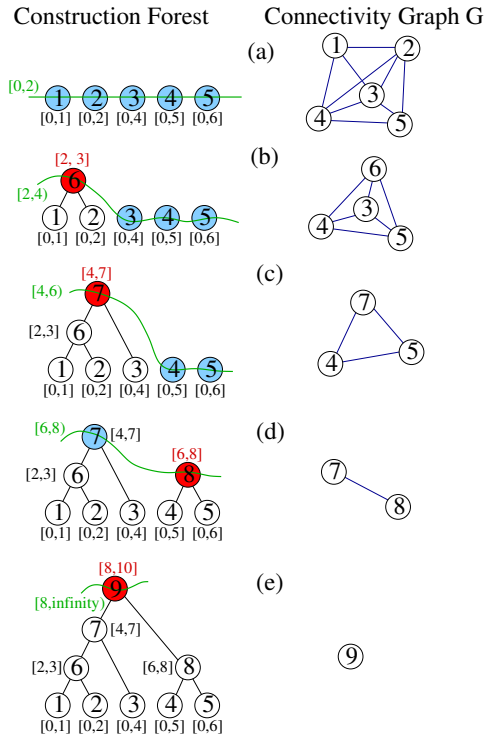

Figure 3: *An example of constructing the merge tree M. The left column shows the construction forest of each stage. Each tree node is associated with an error range* $[\varepsilon_l, \varepsilon_u]$. *Note that the new nodes are created in the order of increasing* ε_l. *From (e), if we query* ε *against the* ε_l *values, as we sweep* ε *from 0 to* ∞, *we see that there are five distinct LOD cuts with their* ε *values falling in five intervals* $[0,2), [2,4), [4,6), [6,8), [8,\infty)$. *These LODs are exactly the cuts (shown in green lines, together with their* ε*-value ranges in green) formed by the root nodes of the construction forests in stages (a)–(e). The right column shows the corresponding connectivity graphs.*

plete; we put x (with key $\varepsilon_u(x)$) to the priority queue Q and repeat the process. Note that since $\varepsilon_l(x)$ of the new node x is always the latest ε_u value extracted from Q, we create the new tree nodes *in the order of increasing* ε_l (see Fig. 3(b)-(e): nodes 6, 7, 8, 9 are created with $\varepsilon_l = 2, 4, 6, 8$).

Right after the internal simplification on x, we obtain the *base mesh* of x and the *refinement sequence* of *vertex splits* for refining the interior of the base mesh. Another sequence, the *simplification sequence* of edge collapses for the boundary, is currently empty and will be grown when other neighbors perform their own global simplification and propagate the affecting edge collapses here. We also maintain an additional, auxiliary *up-to-date* version of the boundary. This is used to provide the boundary information to neighbors that need such data for fold-over checking or other operations.

Properties of the LOD Cuts Referring to Fig. 3, we see that during construction the structure of M starts as a forest, called *construction forest*, which gradually grows into a tree. The "working set" of the nodes are the roots of the forest at each stage, where these roots form a *breadth cut* on tree M. We create a new node at each stage, which has the *largest* ε_l among the cut nodes. Since we create these new nodes in the order of increasing ε_l, these ε_l values form a sorted sequence 2, 4, 6, 8, partitioning the entire error value range $[0,\infty)$ into five intervals $[0,2), [2,4), [4,6), [6,8), [8,\infty)$. From the complete tree M in Fig. 3(e), if we query ε against the ε_l values (i.e., for the highest LOD cut satisfying $\varepsilon_l \le \varepsilon$), then as we sweep ε from 0 to ∞, we see that there are five distinct LOD cuts with their ε values falling in these five intervals; they are exactly the cuts formed by the root nodes of the construction forests in stages (a)–(e). Since at each construction stage we make sure that the cut nodes have consistent boundaries and all possible query LOD cuts were enumerated during construction, any queried LOD cuts are boundary consistent.

Growing the Simplification Sequence Recall that the simplification sequence is grown "passively" by receiving the boundary-edge collapses propagated from affecting neighbors. When such a neighbor ω_1 performs the global simplification up to error $\varepsilon_l(\omega_1)$, all the edge-collapses propagated from ω_1 are ordered sequentially with increasing errors up to $\varepsilon_l(\omega_1)$. We can just accumulate this sequence S_1 with mark $\varepsilon_l(\omega_1)$. Next, the edge-collapses propagated from another neighbor ω_2 have the same property, with sequence S_2 and marked with $\varepsilon_l(\omega_2)$. Recall that we create/simplify tree nodes *in the order of increasing* ε_l, meaning that $\varepsilon_l(\omega_1) \le \varepsilon_l(\omega_2) \le \cdots$, and we can just concatenate $S_1, S_2, \cdots$ sequentially. In the future, to reconstruct the boundary version consistent with ω_i, we just follow the simplification sequence linearly to simplify the boundary up to error $\varepsilon_l(\omega_i)$ (i.e., to the end of S_i), which is very easy.

In Appendix A (in the supplementary materials) we discuss how to support the (extended) quadric error metric [GZ05].

3.3 Run-Time Phase: Out-of-Core LOD Volume Rendering

In the run-time phase, we first load the merge tree M to main memory. Given a user-specified query and the rendering parameters, we perform the following tasks.
(1) Use the tree M and its sub-volumes to find the desired LOD mesh.
(2) Perform volume rendering on the selected LOD mesh.

3.3.1 Task 1: Selecting the Desired Crack-Free LOD Mesh

We support two types of LODs: the *uniform* LODs and the *variable* LODs for selective refinement.

Uniform LODs
We first consider the *uniform* LOD mesh: Given a user-specified error bound ε, we want to find the crack-free LOD

mesh satisfying ε. This is the basis of our algorithm, and will be extended later to support *variable* LOD meshes.

We first find the LOD cut nodes in M satisfying ε based *solely* on ε_l: starting from the root, we perform a breadth-first search to find the *highest breadth cut* (i.e., closest to the root) on M such that each node in the cut has $\varepsilon_l \le \varepsilon$.

Secondly, for each node in the cut, we read the corresponding sub-volume V one at a time from disk to main memory; this includes the base mesh, the refinement sequence, and the simplification sequence. For each V, we do the following.

At the beginning, we use the refinement sequence to refine the interior of the base mesh to satisfy ε: if $\varepsilon_u \le \varepsilon$, then the base mesh already satisfies ε. Otherwise we sequentially refine the base mesh following the refinement sequence, which monotonically decreases the mesh error, until the mesh error is no larger than ε. Let V' be the resulting mesh; note that V' has the same boundary as the base mesh.

Our next task is to use the simplification sequence to simplify the boundary of V' so that the LOD cut on tree M is *crack-free*. Recall from Section 3.2 that the final simplification sequence is a concatenation of marked sequences $S_1, S_2, \cdots$, where each marked sequence S_i has simplification errors no more than $\varepsilon_l(\omega_i)$. In addition, we have $\varepsilon_l(\omega_1) \le \varepsilon_l(\omega_2) \le \cdots$. The operation here is simple: we apply these marked sequences $S_1, S_2, \cdots$ sequentially to simplify V', each time using up the entire marked sequence, until finally we encounter some S_i whose mark is $\varepsilon_l(\omega_i) > \varepsilon$. This means that ω_i is *not* in the LOD cut and we should stop there. Let V'' be the resulting mesh; we call V'' the resulting sub-volume (obtained from the sub-volume V).

Lemma: The selected LOD mesh formed by the resulting sub-volumes V'' is crack-free.

Proof: See Appendix B (in the supplementary materials).

Variable LODs for Selective Refinement

Now we consider the selective refinement queries based on the query (ε, R) where the selection method R is either $R = t\%$ (view-dependent selection) or $R = [a,b]$ (scalar-range query), to be explained soon. Given (ε, R), we want to find the LOD mesh with the highest (i.e., closest to the root) LOD cut possible in which the *active* sub-volumes selected by R are in resolutions satisfying ε and other sub-volumes are in the lowest resolutions just enough to ensure crack-free. Our algorithm consists of the following steps.

1. Use ε to find the LOD cut on the tree M satisfying ε.
2. Among the nodes in the cut, find the nodes selected by R. For $R = [a,b]$, we select the nodes whose $[\min, \max]$ interval[†] intersects with $[a,b]$. For $R = t\%$, we select the closest $t\%$ nodes to the viewer, where currently we estimate the "closeness" by the distance between the viewer and the center of the axis-aligned bounding box of the node's sub-volume.
3. Now we want to adjust the LOD cut to the highest possible while still going through the selected nodes and still being a valid crack-free cut. The intention is to reduce the number of nodes in the cut. For example, in Fig. 3, to select node 3 we can move the cut from the one in Fig. 3(a) up to the one in Fig. 3(b) (but not the one in Fig. 3(c)). To do this, for each selected node we look at its parent's ε_l value. Take the minimum among these ε_l values, and call it ε_{new}, i.e., $\varepsilon_{\text{new}} = \min\{\varepsilon_l(p) | p \text{ is a parent of a selected node}\}$. Recall from Fig. 3 that each LOD cut is associated with a semi-open interval $[e_1, e_2)$; we want to find an LOD cut with a query error bound just a bit smaller than ε_{new} so that the cut is as high as possible but still does not go up to the parent. Thus we set the final query error $\varepsilon_{\text{final}}$ to be $\varepsilon_{\text{new}} - \delta$ for a very small δ value (e.g., 10^{-5}).
4. Use $\varepsilon_{\text{final}}$ to find the LOD cut on the tree M satisfying $\varepsilon_{\text{final}}$. This cut still goes through the selected nodes but goes as high as possible on the other nodes.
5. In the new LOD cut, use the above technique for uniform LOD to simplify the boundary of each node in the cut as necessary to ensure crack-free, but modify the interior refinement: for each selected node we refine the interior up to satisfying ε, and for each of the remaining nodes we just use the *base-mesh interior* without any refinement.

The above scheme is quite flexible in supporting different selection methods R. The proposed view-dependent and scalar-range queries are very effective; see Section 4.

3.3.2 Task 2: Volume Rendering on the Selected LOD Mesh

Our technique for volume simplification and crack-free LOD meshes is general enough for any volume renderer, while our actual volume rendering algorithm makes use of the *HAVS* volume rendering engine [CICS05]. Here we only review the tasks of HAVS that we need to change in order to apply it in our out-of-core setting; for other details we refer to [CICS05]. HAVS needs to extract triangle faces and then do visibility sorting. This extraction requires all vertex and cell information in main memory plus auxiliary data structures. When the mesh is large, it is not affordable to do this.

Here we use a triangle buffer B_t of *fixed* size that can fit a constant number K of triangles. Because our sub-volumes are designed to be small enough to fit in main memory, we can load the sub-volumes in a desired *cut* one by one and extract triangles into the buffer. If all triangles can fit in the buffer then we use the in-core approach, sorting the triangles in-core and perform HAVS rendering. Otherwise, the buffer is written to disk when full. After reading through all sub-volumes, there is a file containing triangles that are organized by sub-volumes, where there are duplicated trian-

[†] For each node of the merge tree M we store the $[\min, \max]$ scalar values of the *most detailed* version of the sub-volume (right before the *internal* simplification begins, and right after the *global* simplification finishes if non-leaf).

Mesh	Fighter	Torso	SF1-x	F16-x
No. verts (K)	257	1,252	16,442	32,853
No. tetras (K)	1,404	4,331	55,921	101,531
No. faces (K)	2,849	8,665	112,084	203,218
File size	47.9MB	147.7MB	2.11GB	3.9GB
HAVS mem	381MB	1.4GB	13.1GB	$>$ 13.1GB

Table 1: *Mesh characteristics.*

gles due to shared triangles between sub-volumes. We perform an external sorting (i.e., out-of-core sorting) on the triangles by the viewing distances, which carries out the desired visibility sorting, with duplicated triangles put together. We then scan the file to remove the duplicates. (For subsequent view-direction changes, the visibility sorting does not involve any duplicated triangles.) Because most triangles are spatially close to each other, the sorting is efficient. Finally we perform a *progressive* volume rendering similar to that in [CBPS06] (albeit not in the client-server mode): for every K triangles we call the display function to render a frame, where in each frame we composite the new image with the previous frame. In this way, we greatly reduce the main memory requirement, and also have the nice feature of progressive volume rendering.

Our resulting volume rendering method can be used as a simple out-of-core volume rendering engine for single-resolution tetrahedral meshes; we call it out-of-core HAVS.

4 Results

We have implemented our technique in C/C++[‡] and ran our experiments on a Dell Precision PC with 1.5GB of RAM, two 3GHz Intel Xeon CPUs, Nvidia GeForce 9800 GTX graphics (512MB graphics memory), and 300GB SCSI 10K rpm disk, running under Fedora-9 64bit Linux OS. The datasets used are listed in Table 1; they are real-world datasets from scientific applications and have been widely used in the visualization research community. Note that tetrahedral meshes need much more information beyond the original input to perform volume rendering. In Table 1, we show the memory footprints of running the original in-core HAVS on these (original) inputs without any LOD structure; for SF1-x the footprint is already 13.1GB.

Simplification

For the preprocessing phase, we show the results of running our out-of-core simplification approach in the top part of Table 2. In Task 1, we partitioned the input mesh into L sub-volumes corresponding to the merge tree leaves. We chose the same number (20K) of vertices in the initial leaf sub-volumes for all datasets, which in turn decided L. We see that the merge tree M and the connectivity graph G were both very small and can easily fit in main memory. Recall that the internal simplification stops when there are no more than c vertices remaining (or when no simplification is allowed); we set c as 1.5K. The overall simplification degree is shown by "No. faces left", which is the ratio of the number of faces between the root sub-volume and the input mesh.

Mesh	Fighter	Torso	SF1-x	F16-x
No. tree nodes	34	92	1187	1761
No. leaves	18	50	624	990
Tree size (KB)	1.6	4.3	55.6	82.5
Max graph size (KB)	1.4	3.8	50.2	77.6
No. faces left (%)	0.94	0.9	0.22	0.0043
Shared verts (%)	21.4	15.3	16.57	27.6
File size (GB)	0.075	0.235	2.6	4
Size increase (%)	57	59.1	23.2	2.5
Scratch space (GB)	0.053	0.252	2.6	5.5
Partition time (s)	30.66	118.27	0.49h	0.92h
Simp. time (s)	90.13	295.57	1.34h	3.53h
Total time (s)	120.79	413.84	1.83h	4.46h
Mem. usage (GB)	0.113	0.202	0.517	1.2
Simp. time (s)	72.89	512.5	N/A	N/A
No. tetra. left (%)	0.94	0.90	N/A	N/A
Mem. usage (GB)	0.582	1.8	N/A	N/A

Table 2: *Simplification results with **1.5GB** of RAM. The top part shows out-of-core results; the bottom part shows in-core results. In the top part, "Max graph size" means the maximum size of the connectivity graph G during simplification. "Scratch space" means the additional disk scratch space beyond the "File size".*

For the purpose of comparison, we also implemented an in-core simplification approach, which reads in the input mesh, performs the edge-collapse simplification with fold-over checking, and stores the base mesh and the refinement sequence, using the same implementation as our internal simplification of sub-volumes. The results are shown in the bottom part of Table 2, where we set the program to stop when reaching the same numbers of vertices as the out-of-core simplification. We see that for Torso a slight thrashing already occurred, resulting in a slower simplification than the out-of-core method (512.5s vs. 295.57s). Typically the memory footprint of the in-core approach is about 12 times the input size, which is certainly too large to handle for SF1-x and F16-x. Clearly, our out-of-core simplification has a significant advantage over the in-core method.

Uniform LOD Rendering

For the run-time LOD rendering, we first tested uniform LODs. We set ε to different values and ran our algorithm; the results are shown in Fig. 4. The corresponding resulting images for most datasets are shown in Fig. 6 (in the supplementary materials). The LOD resolution used in the table is defined as follows. For $\varepsilon = 0$, the LOD resolution is 100%. For other ε values, the LOD resolution is the ratio of the numbers of faces between this LOD and the LOD of

[‡] For the HAVS volume rendering [CICS05] we used the code from the authors: `http://havs.sourceforge.net/`.

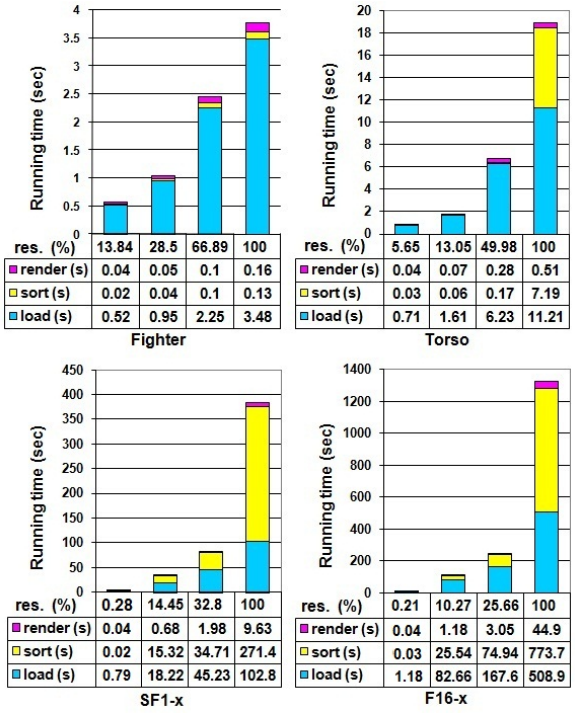

Figure 4: *Running times of our out-of-core volume rendering using* uniform *LODs; "res." means resolution. All images are of size 512 × 512.*

Mesh	Fighter	Torso	SF1-x	F16-x
LOD resolution	66.89%	49.98%	32.80%	25.66%
No. cut nodes	12	42	192	245
OOC LOD				
Total time (s)	2.27	6.78	81.92	245.61
Mem. usage (MB)	313	491	336	382
In-Core LOD				
Total time (s)	2.43	6.83	$>$ 6h	N/A
Mem. usage (GB)	0.31	0.62	3.6	N/A
LOD resolution	100%	100%	100%	100%
No. cut nodes	18	50	624	990
OOC LOD				
Total time (s)	3.77	18.91	383.78	1327.51
Mem. usage (MB)	369	446	629	870
OOC HAVS				
Total time (s)	3.7	18.78	378.2	1182.33
Mem. usage (MB)	381	443	630	870
In-Core HAVS				
Total time (s)	8.1	21.73	$\gg$ 24h	N/A
Mem. usage (GB)	0.38	1.4	13.1	$>$ 13.1

Table 3: *Rendering results using* uniform *LODs with **1.5GB** of RAM for **both** out-of-core (top part) and in-core (bottom part) methods. Out-of-core HAVS (under the same RAM size) is also compared for single-resolution meshes. "OOC" means out-of-core. All images are of size 512 × 512.*

$\varepsilon = 0$. We set the triangle buffer size to be about 100MB. The loading time ("load") in Fig. 4 is from query to obtaining the LOD faces. Recall that the visibility sorting is in-core if the triangle buffer is big enough and out-of-core otherwise. "Sort" in Fig. 4 denotes the visibility sorting time (including the scanning to remove the duplicated faces), and "render" denotes the volume rendering time using HAVS. If the next query has the same ε but different viewing direction, then the total time is the "sort" plus the "render" time where the sorting does not involve duplicates. We can see that the HAVS rendering time is very fast. Observe that the LOD support is essential: for the largest F16-x with more than 101M cells, using 0.21% resolution and after a loading time of 1.18s we can interactively change the viewing direction within 0.07s ($\geq$ 14.28 fps), with image quality not too far from 100% resolution. Also, we can verify that the total running time is proportional to the LOD resolution. Moreover, our memory footprint was at most 870MB and typically much smaller.

For the purpose of comparison, we also tested two in-core methods: (1) in-core LOD, which takes our out-of-core simplification result, reads in the tree and the sub-volumes and keeps them in main memory, and performs the query, where the visibility sorting is in-core sorting and the rendering is directly using HAVS (i.e., non-progressive); and (2) in-core HAVS, which is the original in-core HAVS on the *original input* mesh, *without* using any LOD data structure, for the 100% (single) resolution rendering. For the latter setting, we also compared with (3) out-of-core HAVS. In Table 3, we show the results of these methods compared with our out-of-core results of Fig. 4 for the largest two resolutions in each dataset (note that our memory footprints for the other two smaller resolutions were smaller and are not shown). We see from Table 3 that our advantage of being out-of-core is obvious. As mentioned before, in-core HAVS needs to produce much more information beyond the original input (such as extracting all triangle faces and producing other information), causing a large memory footprint and thrashing. For 100% resolution, out-of-core HAVS was faster than our out-of-core LOD because it worked on a single mesh rather than sub-volumes with duplicated boundaries.

Selective Refinement LOD Rendering

Finally, we ran our out-of-core algorithm and tested selective refinement LOD rendering using queries (ε, R), with $R = t\%$ (view-dependent) and $R = [a,b]$ (scalar-value range), on the two largest datasets. We used a quick, low-quality *uniform* LOD rendered image as shown in the left column of Fig. 5 to guide the selection of R. For SF1-x, this image was rotated to see the color layers for choosing R (see the supplementary short video clip showing the interaction with 3.56% of LOD resolution, where zooming in/out is faster than rotation as sorting is not needed). We show the results in Table 4 and Fig. 5. Since the running time is proportional to the number of LOD faces (as verified from Fig. 4), here we define the

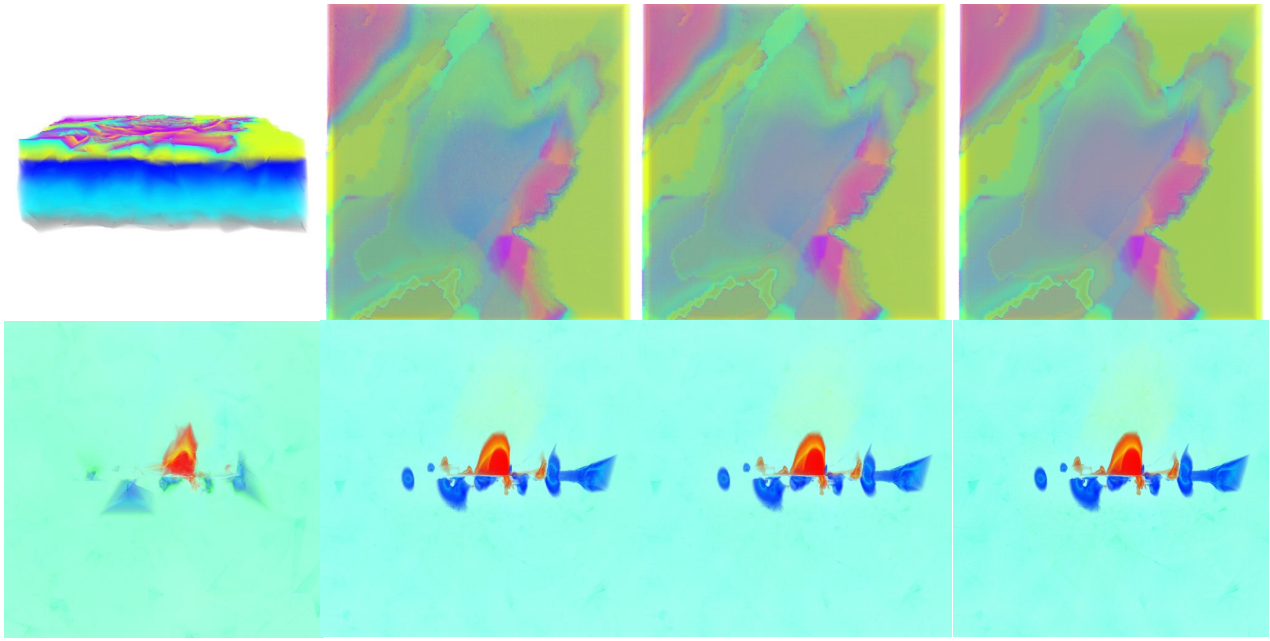

Figure 5: *Representative images of our out-of-core* selective-refinement *LOD volume rendering. Top row: SF1-x; bottom row: F16-x. The left column is the quick, low-quality* uniform *LOD image used to guide the selection of R. The next two columns to the right are for view-dependent selection and scalar-value range queries respectively, both with* $\varepsilon = 0$*; they correspond to the queries in Table 4. The rightmost column shows the full resolution images for the purpose of comparing the image qualities.*

Method ($\varepsilon = 0$)	**view-dependent**		**scalar-range**	
Mesh	SF1-x	F16-x	SF1-x	F16-x
Avg. LOD res.	58.12%	38.56%	70.05%	47.8%
No. sel. nodes	80	60	102	53
No. cut nodes	375	336	491	529
No. leaves	624	990	624	990
Total time (s)	102.36	229.15	125.84	270.70

Table 4: *Out-of-core rendering results using* selective refinement *LODs with **1.5GB** of RAM. Meanings of some entries: Ave. LOD res.:* Average *LOD resolution; No. sel. nodes: number of selected nodes in the LOD cut. All images are of size 512 × 512.*

average LOD resolution to be the ratio between the number of the resulting LOD faces and the number of the faces from the input mesh (i.e., full resolution). In fact the *LOD resolution* defined in Fig. 4 is the same *average* LOD resolution for *uniform* LODs. Here we let $\varepsilon = 0$, i.e., the full resolution was applied to only the *selected* feature portions. Note that since $\varepsilon = 0$, the initial (uniform) LOD cut would go through all leaves, but we can see from Table 4 that our new LOD cut went higher and had much fewer cut nodes. Moreover, since we just selected a small number of nodes and all unselected nodes in the cut used only the *base-mesh interior* with no refinement, our average LOD resolution was further greatly reduced. Comparing the corresponding images (middle two columns vs. the rightmost column in Fig. 5), we see that they are almost of the same image quality, but our new average LOD resolution resulted in a much faster running time. For F16-x the running time was improved from 22.13 minutes (1327.51s) to 3.82 minutes (229.15s, view-dependent), and for SF1-x the improvement was from 6.4 minutes (383.78s) to 1.71 minutes (102.36s, view-dependent), while still retaining the best image quality. Comparing with out-of-core HAVS, we also improved from 19.71 minutes (1182.33s) to 3.82 minutes (view-dependent) for F16-x, showing a huge advantage of our out-of-core LOD approach.

5 Conclusions

We have presented a novel out-of-core simplification and crack-free LOD volume rendering algorithm for tetrahedral meshes. Our experiments showed that we achieve significant speed-ups in both simplification and volume rendering. Although our current focus is on volume rendering, our technique is readily applicable to out-of-core LOD isosurface extraction as well. We plan to extend our technique along this direction to make it a unified out-of-core LOD volume visualization approach for tetrahedral meshes.

Acknowledgments
We thank Cláudio Silva and Steven Callahan for the HAVS code and the test datasets used in this work.

References

[CBPS06] CALLAHAN S., BAVOIL L., PASCUCCI V., SILVA C.: Progressive volume rendering of large unstructured grids. *IEEE Trans. Vis. Comput. Graph. 12*, 5 (2006), 1307–1314. Special Issue for Visualization '06.

[CCSS05] CALLAHAN S., COMBA J., SHIRLEY P., SILVA C.: Interactive rendering of large unstructured grids using dynamic level-of-detail. In *Proc. IEEE Visualization* (2005), pp. 199–206.

[CFM*04] CIGNONI P., FLORIANI L. D., MAGILLO P., PUPPO E., SCOPIGNO R.: Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Trans. Visualization and Computer Graphics 10*, 1 (2004), 29–45.

[CFSW01] CHIANG Y.-J., FARIAS R., SILVA C., WEI B.: A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *Proc. Sympos. Parallel and Large-Data Visualization and Graphics* (2001), pp. 59–66.

[CGG*04] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Trans. Graph. 23*, 3 (2004), 796–803. Special Issue for SIGGRAPH '04.

[CGG*05] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Batched multi triangulation. In *Proc. IEEE Visualization* (2005), pp. 207–214.

[CICS05] CALLAHAN S., IKITS M., COMBA J., SILVA C.: Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Trans. Visualization and Computer Graphics 11*, 3 (2005), 285–295.

[CL03] CHIANG Y.-J., LU X.: Progressive simplification of tetrahedral meshes preserving all isosurface topologies. *Computer Graphics Forum 22*, 3 (2003), 493–504. Special Issue for Eurographics '03.

[CM02] CHOPRA P., MEYER J.: Tetfusion: An algorithm for rapid tetrahedral mesh simplification. In *Proc. IEEE Visualization* (2002), pp. 133–140.

[CRMS03] CIGNONI P., ROCCHINI C., MONTANI C., SCOPIGNO R.: External memory management and simplification of huge meshes. *IEEE Trans. Vis. Comput. Graph. 9*, 4 (2003), 525–537.

[CSS98] CHIANG Y.-J., SILVA C. T., SCHROEDER W. J.: Interactive out-of-core isosurface extraction. In *Proc. IEEE Visualization* (1998), pp. 167–174.

[DDPS05] DANOVARO E., DE FLORIANI L., PUPPO E., SAMET H.: Multi-resolution out-of-core modeling of terrain and geological data. In *Proc. ACM Intl. Sympos. Advances GIS* (2005), pp. 143–152.

[DM02] DE FLORIANI L., MAGILLO P.: Multiresolution mesh representation: Models and data structures. *Multiresolution in Geometric Modelling (M. Floater, A. Iske, E. Quak, editors)* (2002), 363–418. Springer-Verlag.

[FS01] FARIAS R., SILVA C.: Out-of-core rendering of large unstructured grids. *IEEE Computer Graphics & Applications 21*, 4 (2001), 42–51.

[GZ05] GARLAND M., ZHOU Y.: Quadric-based simplification in any dimension. *ACM Trans. Graphics 24*, 2 (2005), 209–239.

[ILGS06] ISENBURG M., LINDSTROM P., GUMHOLD S., SHEWCHUK J.: Streaming compression of tetrahedral volume meshes. In *Proc. Graphics Interface* (2006), pp. 115–121.

[LRC*02] LUEBKE D., REDDY M., COHEN J., VARSHNEY A., WATSON B., HUEBNER R.: *Level of Detail for 3D Graphics*. Morgan-Kaufmann, 2002.

[MHDH07] MUIGG P., HADWIGER M., DOLEISCH H., HAUSER H.: Scalable hybrid unstructured and structured grid raycasting. *IEEE Trans. Vis. Comput. Graph. 6*, 13 (2007), 1592–1599.

[SCCB05] SILVA C., COMBA J., CALLAHAN S., BERNARDON F.: A survey of GPU-based volume rendering of unstructured grids. *Brazil. J. Theo. Appl. Comput. (RITA) 12*, 2 (2005), 9–29.

[SG98] STAADT O., GROSS M.: Progressive tetrahedralizations. In *Proc. IEEE Visualization* (1998), pp. 397–402.

[SM05] SANDER P., MITCHELL J.: Progressive buffers: View-dependent geometry and texture LOD rendering. In *Proc. Symp. Geometry Processing* (2005), pp. 129–138.

[SS06a] SONDERSHAUS R., STRAßER W.: Segment-based tetrahedral meshing and rendering. In *Proc. GRAPHITE '06* (2006), pp. 469–477.

[SS06b] SONDERSHAUS R., STRAßER W.: View-dependent tetrahedral meshing and rendering using arbitrary segments. *J. of WSCG 14* (2006).

[THJ99] TROTTS I., HAMANN B., JOY K.: Simplification of tetrahedral meshes with error bounds. *IEEE Trans. Visualization and Computer Graphics 5*, 3 (1999), 224–237.

[THJW98] TROTTS I., HAMANN B., JOY K., WILEY D.: Simplification of tetrahedral meshes. In *Proc. IEEE Visualization* (1998), pp. 287–295.

[VCL*07] VO H., CALLAHAN S., LINDSTROM P., PASCUCCI V., SILVA C.: Streaming simplification of tetrahedral meshes. *IEEE Trans. Visualization and Computer Graphics 13*, 1 (2007), 145–155.

[VCS*07] VO H., CALLAHAN S., SMITH N., SILVA C., MARTIN W., OWEN D., WEINSTEIN D.: iRun: Interactive rendering of large unstructured grids. In *Proc. Symp. Parallel Graphics and Visualization* (2007), pp. 93–100.

[YL06] YOON S.-E., LINDSTROM P.: Mesh layouts for block-based caches. *IEEE Trans. Vis. Comput. Graph. 12*, 5 (2006), 1213–1220. Special Issue for Visualization '06.

[YSGM04] YOON S.-E., SALOMON B., GAYLE R., MANOCHA D.: Quick-VDR: Interactive view-dependent rendering of massive models. In *Proc. IEEE Visualization* (2004), pp. 131–138.