Efficient Implementation of Real-Time View-Dependent Multiresolution Meshing

Renato Pajarola, *Member*, *IEEE Computer Society*, and Christopher DeCoro, *Student Member*, *IEEE*

Abstract—In this paper, we present an efficient (topology preserving) multiresolution meshing framework for interactive level-of-detail (LOD) generation and rendering of large triangle meshes. More specifically, the presented approach, called FastMesh, provides view-dependent LOD generation and real-time mesh simplification that minimizes visual artifacts. Multiresolution triangle mesh representations are an important tool for reducing triangle mesh complexity in interactive rendering environments. Ideally, for interactive visualization, a triangle mesh is simplified to the maximal tolerated visible error and, thus, mesh simplification is view-dependent. This paper introduces an efficient hierarchical multiresolution triangulation framework based on a half-edge triangle mesh data structure and presents optimized implementations of several view-dependent or visual mesh simplification heuristics within that framework. Despite being optimized for performance, these error heuristics provide conservative error bounds. The presented framework is highly efficient both in space and time cost and needs only a fraction of the time required for rendering to perform the error calculations and dynamic mesh updates.

Index Terms—Level-of-detail, multiresolution modeling, mesh simplification, interactive rendering.

1 Introduction

Despite alternative representation and rendering approaches, polygon meshes are still the standard in interactive visualization systems such as VR environments, real-time simulations, and interactive 3D games. Increasingly complex polygonal models exist today ranging from detailed CAD/CAM representations, high-resolution isosurface extractions from volume data sets, extensive terrain surface models, large virtual environments, to extremely complex digitized models such as the statues from the Digital Michelangelo Project [1]. Such large models are hard to render at interactive frame rates due to the very high number of polygons that exceeds the per-frame rendering capacity even of high-end graphics hardware. Level-of-detail (LOD) based visualization techniques as proposed in [2] or [3] allow rendering of the same object using several different triangle meshes of variable complexity. Thus, the mesh complexity can be adjusted according to the object's relative position from the viewer, its visual importance in the rendered scene, and with respect to the available rendering power to guarantee stable interactive frame rates. Many mesh simplification and multiresolution triangulation methods have been developed to create different LODs, sequence of LOD-meshes with increasing complexity, and hierarchical triangulations for LOD-based rendering (see also overviews [4], [5], [6], [7]). LOD methods not only reduce the time used to perform geometric transformations, lighting calculations, and rasterization on the graphics hardware, they also contribute to reducing the bandwidth bottleneck of transferring large amounts of data from main memory to the graphics card.

Ideally, for rendering, the size of a triangle mesh representing a surface or object is reduced to the maximal error that can be visually distinguished or tolerated on screen by the user. While finding this minimal triangle mesh representation is a very hard problem and much too time-consuming to do in real-time, the following heuristics can be used to guide the mesh simplification efficiently (see also Fig. 1 for examples):

1. Parts of the surface outside of the view area can be simplified. This reduces the number of vertices processed by the rendering pipeline for *view-frustum culling*.
2. Invisible surface areas oriented away from the viewer can be simplified. This reduces the amount of work for *back-face culling* in the rendering pipeline.
3. *Silhouette* regions should be preserved due to their visual importance.
4. Surface regions with a very small projected area on screen may be represented with only a few triangles. Simplification can be performed until a user specified *screen-projection* area tolerance is reached.
5. Sufficiently planar regions with equal surface normals can be represented with few coplanar triangles. Simplification can be performed until a user specified tolerance in surface-normal deviation, or *shading* tolerance is reached.

Real-time rendering of a triangulated surface exhibits an extremely high frame-to-frame coherence of the displayed triangles. If a triangle has been rendered in one frame, it is very likely that it also has to be rendered in the next frame since the viewpoint and view-directions change smoothly

- *R. Pajarola is with the Computer Graphics Lab, Information and Computer Science, University of California, Irvine, CA 92697-3425. E-mail: pajarola@acm.org.*
- *C. DeCoro is with the Center for Advanced Technology, Courant Institute of Mathematical Sciences, New York University, 719 Broadway, Suite 1210, New York, NY 10003. E-mail: cdecro@cat.nyu.edu.*

Manuscript received 14 Sept. 2001; revised 25 Aug. 2002; accepted 14 Nov. 2002.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 114980.

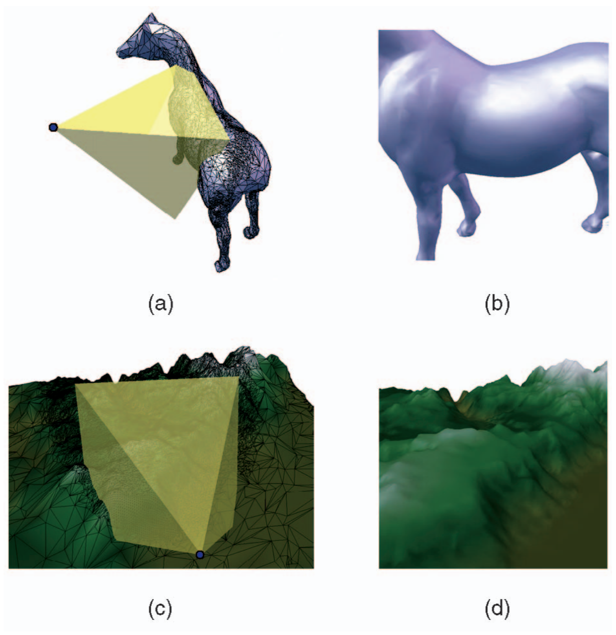


Fig. 1. View-dependent meshing examples. (a) View of the view-frustum and simplified mesh (rendered 9,404 out of 96,966 faces). (b) Same triangle mesh viewed from the actual viewpoint. (c), (d) View-dependent meshing of terrain data (rendered 41,768 out of 178,802 faces).

over time and only very little from one frame to the next. Therefore, to take advantage of processing only the minimal number of mesh elements, the triangle mesh has to be updated incrementally from frame to frame based on the currently rendered set of triangles. Instead of performing mesh simplification on the entire mesh every time the view has changed, simplification and refinement operations can be performed on the current mesh if one of the surface properties 1 through 5 listed above changed locally—meaning locally on the triangulated surface. In fact, the temporal coherence and incremental update make mesh simplification according to properties 1 and 2 really useful.

While a lot of work has been done on mesh simplification for geometric approximation, fewer approaches have addressed the problem of view-dependent simplification of arbitrary triangle meshes for real-time rendering and performance optimization was not the main focus of most approaches. In this paper, we present an efficient implementation of a view-dependent and topology preserving meshing framework called FastMesh that has optimized data structures and algorithms for maintaining a dynamically simplified and refined mesh, uses minimal amount of storage, has compact data structures exploiting implicit relations between different elements, and exhibits short preprocessing time. Furthermore, we present an optimized implementation for the five view-dependent simplification criteria listed above with very low CPU time consumption. While the presented FastMesh framework provides minor conceptual advancements in view-dependent mesh simplification in general, it does provide major performance improvements in terms of efficient algorithms, data structures, and numerical error calculations, as well as showing implementation details for view-dependent meshing. In this context, the two main contributions of this paper are:

- an efficient implementation of a hierarchical, half-edge data structure based multiresolution triangulation framework for view-dependent LOD rendering,
- optimized computation and implementation of view-dependent error heuristics for mesh simplification with conservative error bounds.

The remainder of the paper is organized as follows: In Section 2, we discuss preliminary technical details and we discuss and compare previous related work on view-dependent mesh simplification. Section 3 introduces the data structures and algorithms to handle the dynamically changing triangle mesh and Section 4 describes the visual error heuristics. In Section 5, we provide details on the initialization of a FastMesh data structure and implementation-specific details as well. Experimental results are given in Section 6 and Section 7 concludes the paper with a summary of the results and future work.

2 Related Work

An abstract description of the basic FastMesh view-dependent meshing framework was presented in [8]. This paper provides an extended overview on related work, increased details on the data structures and algorithms, improvements and additions to the visual error heuristics, explanations on retaining correct boundary conditions, some discussion on nonmanifold triangle meshes, and, in particular, it presents a new and significantly more space efficient implementation of the proposed data structures. In this section, we want to review the most relevant work on mesh simplification with respect to our approach and other previously developed view-dependent mesh simplification approaches.

2.1 Mesh Simplification

Numerous methods for mesh simplification have been developed in the last decade and a discussion of them is beyond the scope of this paper. For an overview on the various mesh simplification methods, see [5], [6], or [9]. Here, we want to highlight the work of [10] on progressive simplification of triangle meshes and [11] on efficient geometric error metrics.

In [10], a sequence of n *edge collapse* (ecol) operations is applied to simplify an arbitrary manifold mesh M^n to a simpler mesh M^0 of the same topology, reducing the number of vertices by n. Given the coarse mesh M^0, i different LOD approximations M^i can be reconstructed by applying a set of i *vertex split* (vsplit) operations—the inverse of the ecol operation—to the base mesh M^0. In FastMesh, we use the half-edge collapse variant shown in Fig. 2 that collapses a directed half-edge v_1v_2 to its endpoint v_2. The use of half-edge collapse operations has also been advocated in [12] and [13] for fast mesh simplification, however, not in the context of generating a high-performance view-dependent multiresolution hierarchy.

To create an initial partially ordered set of ecol simplification operations, we use the *quadric error metric* introduced in [11]. This quadric error metric is based on the fact that the squared distance of a point $v = (x, y, z, 1)$ from a plane $p = [a, b, c, d]^T$ with $a^2 + b^2 + c^2 = 1$ can be

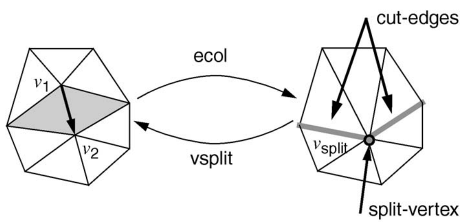


Fig. 2. Half-edge collapse (ecol) and vertex split (vsplit) operations for triangle mesh simplification and refinement.

computed by a matrix multiplication $(p^T \cdot v)^2 = v^T \cdot K_p \cdot v$ using the 4×4 quadric

$$K_p = \left[a^2 \;\; ab \;\; ac \;\; ad,\; ba \;\; b^2 \;\; bc \;\; bd,\; ca \;\; cb \;\; c^2 \;\; cd,\; da \;\; db \;\; dc \;\; d^2\right]$$

for the plane p. Furthermore, the sum of squared distances to a set of planes P can easily be computed as $d(v,P)^2 = \sum_{p \in P} v^T \cdot K_p \cdot v = v^T \cdot \sum_{p \in P} K_p \cdot v$. Thus, a set of planes can be represented by one quadric $Q_P = \sum_{p \in P} K_p$ and merging two disjunct set of planes P_1 and P_2 can be achieved by $Q_{P_1 \cup P_2} = Q_{P_1} + Q_{P_2}$. We would like to note here that the squared distance of v to a set of planes Q as calculated by $v^T \cdot Q \cdot v$ increases directly with the number of planes. This is not a desired effect for geometric approximation since we would like the error metric not to be dependent on the number of planes but rather be defined by their geometric configuration. For example, the distance of a vertex to multiple almost coplanar planes should be roughly the same as the distance to just one of these planes. Therefore, in our implementation, we normalize the coefficients of Q_P by the number of planes $n_P = |P|$ to get the *mean squared distance*. Thus, adding a single plane quadric K to Q_P is done by $Q = (Q_P \cdot n_P + K)/(n_P + 1)$ and adding two quadrics Q_{P_1} and Q_{P_2} results in $Q = (Q_{P_1} \cdot n_{P_1} + Q_{P_2} \cdot n_{P_2})/(n_{P_1} + n_{P_2})$.

The preprocess of selecting a set of ecol operations is initialized by assigning each vertex v a quadric Q_v that is initialized to the set of planes of triangles incident upon v. For a half-edge collapse v_1v_2, the quadrics of the collapsed vertices are added $Q = (Q_{v_1} \cdot n_{v_1} + Q_{v_2} \cdot n_{v_2})/(n_{v_1} + n_{v_2})$ and the geometric approximation error introduced by that edge collapse is estimated by $v_2^T \cdot Q \cdot v_2$. If the half-edge collapse is performed, the quadric Q is assigned to the endpoint v_2 of the half-edge. Note that any other error metric can be used as well to guide selection of good edge collapses. In particular, the one-sided quadric error metric proposed in [13] may be a suitable alternative.

The current implementation of the FastMesh preprocess differs from standard mesh simplification methods not only by the above error metric but also in the order in which edge collapse operations are selected. Instead of selecting a sequence of edge collapses in a greedy manner as many mesh simplification methods do—iteratively selecting the edge collapse with the smallest error next—FastMesh iteratively selects largest independent sets of edge collapses similar to [14] or [15]. This strategy is used to minimize constraints between simplification and refinement operations, see Sections 3 and 5 for more details.

2.2 View-Dependent Meshes

In this section, we want to review the most relevant view-dependent mesh simplification approaches that have previously been proposed and summarize the differences to our approach.

In [16], view-dependent mesh simplification was introduced based on the basic ecol and vsplit operations as outlined above. A sequence of edge collapse operations is selected based on the edge length as geometric error metric which defines a binary hierarchy on the vertices. At runtime, a front through the vertex hierarchy separates the vertices that define the current mesh (in and above the front) from the removed vertices (below the front) and is updated for each frame. While other criteria are discussed in [16], only the projected length of collapsed edges is implemented to adjust the vertex front at runtime. Surface normal information is additionally incorporated in [17]. The extension in [18] allows to incorporate topology simplification and handling of nonmanifold meshes through the use of virtual edges.

A similar approach is presented in [19], where a sequence of ecol operations as proposed in [10] is used to construct the binary vertex hierarchy. Furthermore, two new image-space visibility heuristics are given and a screen-space geometric approximation error metric is proposed to update the vertex front. In [20], the approach is improved in particular with respect to main memory space cost at the expense of increased complexity of the data structures and continuous memory allocation and deallocation at runtime.

Also based on a binary vertex hierarchy defined by a sequence of edge collapse operations is the approach in [21]. However, unlike the previous methods, it supports the unconstrained space (simply connected, nondegenerate, manifold) of all possible meshes that can be generated by a given partially ordered set of edge collapse operations.

Another view-dependent triangulation approach called *Multi-Triangulation* (MT) based on vertex insertion and removal is proposed in [22] and improved for storage cost and client-server interaction in [23] and [24]. A partially ordered set of vertex insertions is maintained in a direct acyclic graph (DAG) that allows for selective view-dependent mesh refinement. A cut through that dependency graph, similar to a front in a vertex hierarchy as in the methods above, represents a particular mesh at runtime. While efficient in terms of space cost, the MT update operations are more costly than the vertex split and edge collapse operations. No novel view-dependent mesh simplification heuristics are defined or efficient implementations thereof are proposed.

A generalized framework on the basis of vertex hierarchies created from any kind of vertex contraction is presented in [25]. This approach is very general, but it uses much more storage than others and does not provide highly optimized calculations of view-dependent error heuristics. While the method in [25] can be viewed as the least restricted generalization of view-dependent meshing, our approach targets the other end of high-performance implementation of view-dependent multiresolution meshing. A vertex-clustering hierarchy similar to [25] has also

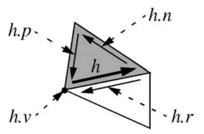


Fig. 3. Half-edge data structure.

been proposed in [26] using simple screen-projection size of vertex clusters to determine expansion of individual nodes for each frame. However, the algorithms and data structures for maintaining the dynamically changing triangle mesh are not very efficient in terms of space and time cost.

Specialized view-dependent triangulation methods have been proposed for regular or pseudoregular meshes. In particular, approaches such as [27], [28], and [29] support efficient view-dependent rendering of large scale terrain meshes. These approaches are commonly based on the screen-projection of a vertical surface displacement introduced by removed vertices. While the computation of such an error metric is very fast, it is not directly applicable to arbitrary manifold meshes. Also, the hierarchical incremental Delaunay triangulation presented in [30] provides some view-dependent error metrics for terrains.

In comparison to the most closely related approaches [16], [19], [21], [22] and their extensions, our approach differs and improves in the following ways:

- more compact and space efficient data structures,
- optimized view-dependent mesh simplification heuristics using only two parameters,
- reduced partial ordering restrictions of edge collapse operations.

3 Dynamic Meshing

In this section, we describe the data structures and algorithms for maintaining a view-dependent mesh using a half-edge data structure [31]. The half-edge data structure is a simplified version of the doubly connected edge list (DCEL) [32] without vertex-to-edge and face-to-edge relations. Moreover, the winged-edge (WE) data structure [33] can be viewed as a pair of two half-edges. Because a vertex split operation introduces new triangles along two cut-edges, it is more natural to use the half-edge instead of the winged-edge representation. Our implementation of a half-edge mesh data structure is further simplified as outlined below and does not require any edge-to-face pointers, in contrast to the DCEL and WE data structures. The use of a half-edge data structure makes the management of variable triangle mesh connectivity very efficient both in space and time cost, as evidenced by the experimental results in Section 6. A similar *directed edge* data structure was proposed in [34] to efficiently maintain a dynamically changing triangle mesh data structure. We compare our experimental results to [34] in Section 6.

We want to stress here that using a half-edge collapse operation for mesh simplification is not equivalent to maintaining the triangle mesh connectivity using a half-edge data structure. The choice of simplification operation and error metric is largely independent of the mesh representation concept. On the other hand, the half-edge

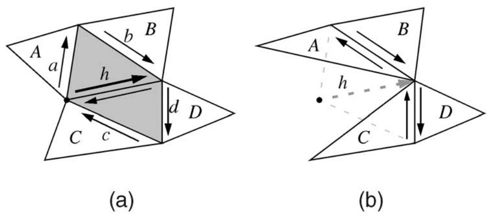


Fig. 4. (a) Half-edge collapse and (b) vertex split.

collapse operation is a natural fit for the half-edge data structure that we propose for dynamic mesh representation.

3.1 Half-Edge Data Structure

In FastMesh, a half-edge data structure stores the connectivity information of the triangle mesh and each triangle face is implicitly represented by an ordered set of three oriented half-edges. Every half-edge h stores its reverse *twin* half-edge $(h.r)$, the next $(h.n)$, and previous $(h.p)$ half-edges of that triangle, and the starting vertex $(h.v)$ of the half-edge as shown in Fig. 3. This mesh representation allows efficient traversal and neighbor finding on the triangulated surface.

FastMesh stores a mesh of m triangles as an array of $3m$ half-edges. Note that we will use h interchangeably to refer either to the integer index or the pointer representation of a half-edge. In the half-edges array, each group of three consecutive half-edges defines a triangle. Thus, the face index f corresponding to a half-edge h is $f = face(h) = h$ DIV 3. The first half-edge h of a face f corresponds to $h = \mathrm{edge}(f) = 3f$ and, in CCW order, the second is $3f + 1$ and the third half-edge is $3f + 2$. Therefore, the previous $h.p$ and next $h.n$ fields of a half-edge h are implicitly given by integer-division (DIV) and modulo (MOD) operations on indices or by a simple conditional statement (but we will keep the notation $h.n$ and $h.p$ for simplicity),

$$h.n = IF(hMOD3 = 2)THEN(h - 2)ELSE(h + 1)$$
$$h.p = IF(hMOD3 = 0)THEN(h + 2)ELSE(h - 1).$$

For a half-edge collapse of edge h, the mesh connectivity has to be updated as shown in Fig. 4. This can be done efficiently using the half-edge mesh data structure. Collapsing the half-edge h and deactivating its incident triangles from the list of rendered triangles only requires updating the reverse information of half-edges a, b as well as c, d such that the triangle pairs A and B as well as C and D become direct neighbors, as shown in Fig. 4b). For triangles A and B, and given the half-edge h to be collapsed, this can be efficiently done by:

$$h.p.r.r = h.n.r \text{ and } h.n.r.r = h.p.r. \tag{1}$$

Two similar assignments are required to set up the reverse information between triangles C and D. Note that the half-edge entries of triangles $face(h)$ and $face(h.r)$ are not altered at this point; these triangles are just deactivated for rendering. The inverse vsplit operation will reuse this information and only the index of the collapsed half-edge h has to be recorded for a vsplit.

In addition to updating the mesh connectivity, all half-edges that have the same start-vertex $h.v$ need their

```
t = h.p.r;              // equal to a
while (t != h.r.n)      // visit edges up to c
   t.v = h.n.v;
   t = t.p.r;           // rotate CCW around h.v
end
```

Fig. 5. Updating start-vertex for half-edge collapse.

start-vertex to be reassigned to the end-vertex of h, which is $h.n.v$. This can be done by rotating (counterclockwise) around the removed vertex $h.v$ and visiting all outgoing half-edges between a to c (see Fig. 4). The code segment shown in Fig. 5 performs this update for a half-edge collapse h.

Given a collapsed half-edge h, a vsplit operation must update the mesh connectivity to include the two triangles originally incident upon h. Note that as, pointed out above, the entries in the half-edge table for these two triangles have not been altered after collapsing h. Therefore, the original connectivity for triangles A and B in Fig. 4 can be restored using the available half-edge table entries for h by:

$$h.p.r.r = h.p \text{ and } h.n.r.r = h.n. \tag{2}$$

Similarly, triangles C and D can be efficiently updated. Furthermore, the start-vertex of all half-edges incident upon the split-vertex $h.v$ between and including triangles A and C—between half-edges a and c in counterclockwise orientation around $a.v$—has to be updated. The start-vertex must be updated from $h.n.v$ to be $h.v$. Note that $h.v$ is indeed the correct start-vertex because the information for h has not been changed. This update can be done analogously to the code example in Fig. 5.

We want to point out here that, for (2) to work in a dynamically changing mesh, some conditions have to be met. In particular, the four faces A, B, C, and D have to be exactly in the configuration as shown in Fig. 4b before the vertex split reversing the edge collapse h can be performed. In Section 3.4, we discuss how vertex split operations can be performed if this condition is not directly met.

3.2 Boundary Conditions

The simplification and refinement operations on the half-edge data structure as defined in the previous section work well for manifold orientable meshes of arbitrary genus without boundary. We will discuss in Section 5.4 how nonmanifold and nonorientable meshes can be handled in practice. Simple boundaries of manifold meshes do not constitute a conceptual problem for the half-edge data structure or edge collapse operations and are discussed here.

A boundary edge h in a half-edge data structure can easily be detected when the reverse pointer $h.r$ is not set (see also Fig. 3) or if it is set to an invalid value. Thus, boundary edges are uniquely defined and any attempted mesh traversal across a boundary edge can safely be stopped when a boundary condition is detected. The most critical mesh traversal with respect to boundary edges is to visit all incident edges of a vertex starting at a given outgoing half-edge h. Visiting incident edges of a boundary vertex v is performed in two steps, as shown in Fig. 6a. Starting at a given half-edge h, the counterclockwise (CCW) rotation $t = t.p.r$ can be stopped when t gets invalid, set to a

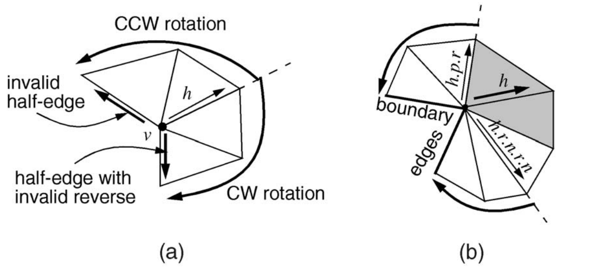


Fig. 6. (a) Visiting all incident edges or vertices around a boundary vertex. (b) Mesh traversal around vertex of collapsed half-edge h.

nonexisting reverse of a boundary edge, and the clockwise (CW) rotation $t = t.r.n$ can be stopped as soon as $t.r$ is invalid. The vertex reassignment of a half-edge collapse h as outlined in the previous section is also performed by two rotations, as shown in Fig. 6b. When collapsing a half-edge h and its incident triangles (gray shaded triangles in Fig. 6), the CCW rotation starts at $t = h.p.r$ and the CW rotation starts at $t = h.r.n.r.n$ (the start-vertex of $t = h.r.n$ does not have to be changed).

Furthermore, also updating the reverse information of the mesh connectivity for a half-edge collapse h has to take care of possible boundary conditions, as shown in Fig. 7. The reverse edge update assignments of (1) must only be performed for $h.n$ or $h.p$ if they are not boundary edges, thus if $h.n.r$ or $h.p.r$ actually point to existing edges. But, in that case, (1) can still be used as is. Thus, in Fig. 7a, we would only update $h.n.r.r = h.p.r$ for the triangle containing h (the other triangle containing $h.r$ is handled normally). In case the collapsed half-edge h itself is a boundary edge, as shown in Fig. 7b, the reverse side $h.r$ is simply not considered for any mesh connectivity updates and vertex reassignments.

The vertex split refinement operation deals with the boundary conditions explained above analogously to the half-edge collapse and a detailed discussion is omitted here. The presented boundary handling solutions apply to any manifold triangle meshes with boundaries, also to holes or internal boundaries. The presented mesh update operations and boundary handling will preserve the topology of the input mesh and holes will not be filled or removed due to the topological constraints given in Section 3.4 and illustrated in Fig. 10.

3.3 Multiresolution Hierarchy

Conceptually, in our approach, a sequence of ecol operations computed during a preprocessing step defines a binary hierarchy similar to [16], [17], [19], [20] and [21] and

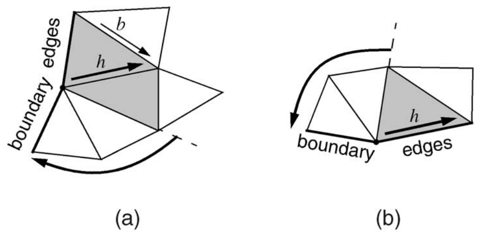


Fig. 7. (a) Reverse half-edge update of boundary edges. (b) Half-edge collapse on boundary.

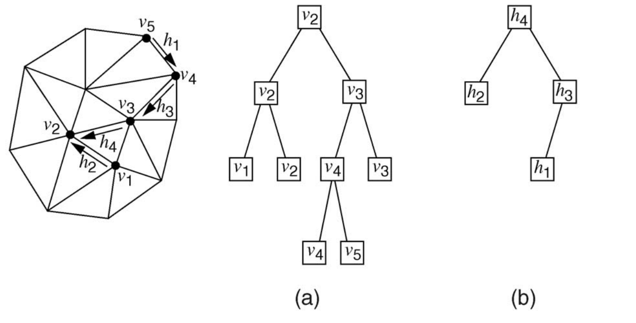


Fig. 8. Binary half-edge collapse hierarchy H. (a) Vertex hierarchy. (b) Half-edge collapse hierarchy H.

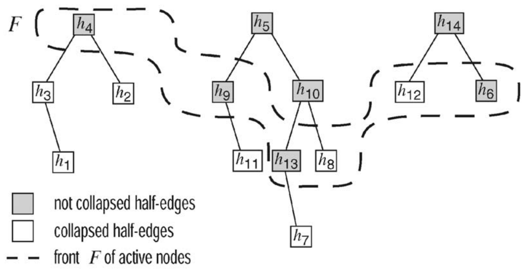


Fig. 9. Front F of the current view-dependent mesh through the binary half-edge collapse hierarchy H.

a view-dependent mesh is defined by a front of active nodes through this hierarchy. However, the hierarchy in FastMesh differs significantly in implementation and semantics from the previous approaches.

In contrast to previous approaches and to reduce storage cost, we define the multiresolution hierarchy H on the half-edges that are collapsed since the leaf nodes of a vertex hierarchy do not carry any information required for collapsing an edge or splitting a vertex. Thus, H requires only half as many nodes as vertex tree-based representations. Furthermore, this half-edge collapse hierarchy, as shown in Fig. 8, is implemented as a separate data structure, not merged with the vertex data itself, and only stores information per node that is required for the view-dependent error heuristics. A node $t \in H$ consists of pointers to the parent node $t.p$, left $t.l$ and right $t.r$ child nodes, and an index $t.h$ for a half-edge collapse of edge h. Additionally, each node t also stores two parameters to compute the visual error heuristics (see also Section 4).

This definition of the binary half-edge collapse hierarchy H completely changes the semantics of the front $F \subset H$ of active nodes through the hierarchy that defines a particular LOD mesh at runtime. Although the front F defines a particular LOD mesh based on which edges are currently collapsed, it does not directly represent all vertices of the current mesh as in the previous approaches (these vertices are only implicitly given by F).

Definition. *In FastMesh the front F consists of all* active *nodes in H, see also Fig. 9. A node t is defined to be active if and only if one of the following two properties holds:*

1. *$t.h$ is currently not collapsed and both child nodes $t.l$ and $t.r$ are either currently collapsed or not existing. (subset F_1 of F),*
2. *$t.h$ is currently collapsed, its parent $t.p$ is not collapsed, and its sibling child node in $t.p$ exists and is not collapsed (subset F_2 of F).*

At any time, F contains a node of every possible path from the roots to the leaves of H, but only one node of any particular path at a time. F can be implemented as a doubly linked linear list. In fact, as described in more detail in Section 3.4, F contains exactly all nodes for the current LOD mesh that can potentially be collapsed (nodes of $F_1 \subset F$) or that must be checked for mesh refinement (nodes of $F_2 \subset F$ and all child nodes of $F_1 \subset F$ given by $C_{F_1} = \{t | p \equiv t.p \wedge p \in F_1\}$).

At runtime, for every change in view parameters, the front F is traversed and updated dynamically according to the view-dependent mesh simplification heuristics described in Section 4. First, the nodes $t \in F_1$ are tested to be collapsed and, second, all nodes $t \in (F_2 \cup C_{F_1})$ are tested to be split. After a collapse operation, the parent node is recursively collapsed if applicable and, after a split operation, the corresponding child nodes are recursively split if necessary. All simplification and refinement operations have to be tested at runtime to be valid as described below since they are performed out-of-order with respect to their partial ordering at initialization.

3.4 Preconditions

Because mesh simplification and refinement operations are not performed in the same order as in the initialization stage for view-dependent meshing, we must carefully examine the conditions under which these updates can be performed.

Partial Ordering. The iterative selection of edge collapse operations at initialization imposes a partial ordering on the mesh simplification and refinement operations which is manifested in the binary hierarchy H. Let us denote the subtree rooted at node $t \in H$ by $H_t \subset H$. A node $t \in H$ can be collapsed, respectively, its half-edge $t.h$, only if all other descendant nodes in H_t correspond to currently collapsed half-edges. On the other hand, a node t can only be split if there exists no other collapsed ancestor node $s \in H$ with $t \in H_s$. In other terms, edge collapse operations must be performed bottom-up in correct partial order.

At runtime, this partial ordering is automatically and efficiently enforced by incrementally expanding or contracting the front F as described in the previous section. For a particular LOD mesh, the nodes in F_1 exactly denote all edge collapse candidates that satisfy the above partial ordering constraint. Similarly, the nodes in $F_2 \cup C_{F_1}$ are exactly all vertex split candidates that satisfy the above partial ordering constraint.

Topology. For any mesh update operation to be valid, it must not change the mesh topology, such as genus or boundary components, and it must not introduce any nonmanifold mesh connectivity. Thus, a half-edge collapse h is considered to be invalid if there exists any vertex V that is adjacent to both endpoints P and Q of h and for which the triple (P, Q, V) is not a triangle in the current mesh. An example of such an invalid configuration is shown in Fig. 10.

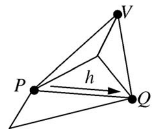


Fig. 10. Topological ecol constraint. Half-edge h cannot be collapsed because of P and Q being connected to V.

This constraint preserves the topology of the input mesh and also prevents minimal boundaries consisting of only three vertices to be reduced further. Note that a temporarily invalid half-edge h may become a valid half-edge collapse again at any other time during view-dependent meshing due to the dynamically changing mesh connectivity. Thus, the validity of a half-edge collapse is not a static attribute and must be checked whenever attempting to collapse a half-edge.

The topological constraint for an edge collapse operation can efficiently be tested at runtime by examining the direct neighbors of the half-edge h. Given the set N_v of vertices adjacent to a vertex v, the topological constraint is satisfied if:[1]

$$N_{h.v} \cap N_{h.n.v} = \{h.p.v, h.r.p.v\}.$$

Besides the topological constraint outlined above, there are no other hard constraints that must be satisfied for a half-edge collapse operation. This is in contrast to the closely related approaches [19] and [16], [17] which require an explicit neighborhood configuration before an edge can be collapsed. Even the implicit dependencies introduced in [18] restrict the ecol operation to a specific configuration of the immediately adjacent vertices. Only the approach in [21] provides the unconstraint simplifications.

To improve the quality of the generated triangle mesh, additional *soft* constraints can be included to prevent triangles with bad aspect ratio or flipping of triangle normals. A half-edge that should be collapsed for the current frame and that does not satisfy the topological or any additional soft constraint is deferred. It is evaluated again after the current frame has been rendered and the mesh is updated again. This cannot be easily avoided with our and similar edge-collapse based approaches and, occasionally, causes a small delay of a simplification operation, but provides conservative view-dependent meshing.

A vertex split operation is defined by the split-vertex v_{split} and two outgoing half-edges e_l and e_r (with $e_l.v = e_r.v = v_{split}$) or, alternatively, by two cut-vertices $v_l = e_l.n.v$ and $v_r = e_r.n.v$, as shown in Fig. 11a. The only topological constraint to split a vertex v_{split} is that v_l and v_r must not be the same vertices ($v_l \neq v_r$). Again, this situation may occur due to the view-dependent mesh simplification which performs edge collapse and vertex split operations in arbitrary conforming partial order as described at the beginning of this section. To guarantee conservative view-dependent meshing, if $v_l = v_r$, then we can recursively check and refine the node $t \in H$ that splits v_l. Besides [21], all other approaches [16], [17], [18], [19] have more restrictive constraints requiring some explicit or implicit neighborhood configuration around the split-vertex.

1. Simplified condition for manifold meshes without boundary given here.

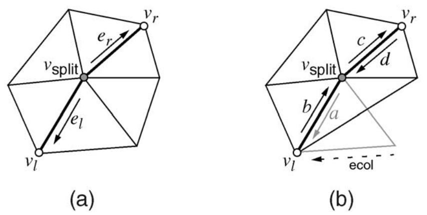


Fig. 11. (a) Valid vertex split with cut-edges e_l, e_r and cut vertices $v_l \neq v_r$. (b) Triangle containing half-edge a has been collapsed.

Ill-Defined Cut-Edges. A vertex split specified by a node $t \in H$ is defined by the corresponding collapsed half-edge $t.h$. As indicated in Section 3.1, the cut-edges e_l and e_r can be recovered by the information stored in the half-edge table for the collapsed half-edge h. Let us here use $a = h.r.p.r$, $b = h.r.n.r$, $c = h.p.r$, and $d = h.n.r$. Thus, in the normal case, the cut-edges can be recovered by $e_l = a$ and $e_r = c$ and it holds that $a = b.r$, $c = d.r$. However, in a dynamically simplified mesh, it can occur that, after collapsing a half-edge h to v_{split}, other simplification operations can remove the triangle face containing a (or b, c, d), as shown in Fig. 11b. In that case, there is a node $s \in H$ corresponding to that half-edge collapse operation with $face(a)$ being either equal to $face(s.h)$ or $face(s.h.r)$. Due to the partial ordering in the binary tree hierarchy, we can enforce the current vertex split by first propagating the split operation to node s such that the required face a is reintroduced into the triangle mesh. This is called a *forced split*.

4 View-Dependent Error Heuristics

In this section, we describe the efficient computation of the view-dependent and visual error heuristics introduced in Section 1 while Appendix I (which can be found on the Computer Society Digital Library at http://computer.org/tvcg/archives.htm) provides details on the efficient calculation of each heuristic. These error heuristics have to be computed for each frame for all active nodes when the front F is traversed, as explained in Section 3.3. The nodes are collapsed or split based on: *view-frustum culling, back-face culling, silhouette preservation, screen-projection tolerance*, and *shading tolerance*. Since the number of times these criteria must be computed per frame is on the order of the number of vertices in the current mesh, their computation must be very efficient. FastMesh's view-dependent error heuristics are designed to minimize computational costs, but nevertheless compute conservative error bounds on all five criteria. The basic ideas of using most of the presented visual error heuristics for view-dependent mesh simplification have been proposed in previous approaches. However, no consideration of efficiency nor an optimized implementation has been given. Besides defining a few additional visual error heuristics that have not been proposed so far, this section and Appendix I (which can be found on the Computer Society Digital Library at http://computer.org/tvcg/archives.htm) mainly focus on the efficient computation and implementation of these simplification criteria.

As mentioned earlier, only two scalar values per node $t \in H$ in the half-edge collapse hierarchy are required as

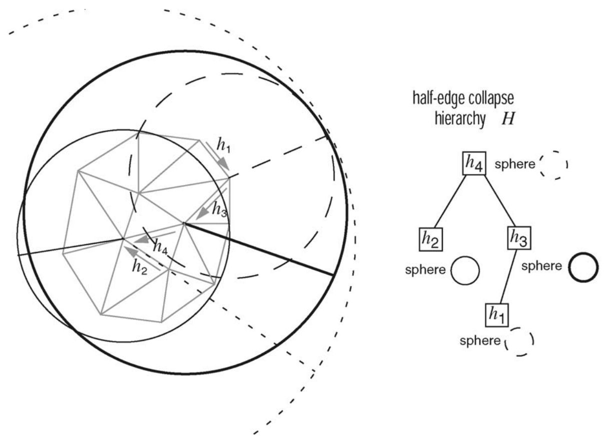


Fig. 12. Bounding spheres of a sequence of half-edge collapse operations.

parameters to compute all five error heuristics. These two parameters are explained below.

Bounding sphere radius. The bounding sphere centered at the endpoint of a half-edge indexed by $t.h$ with radius $t.radius$ encloses all triangles that are affected by the half-edge collapse t and its descendants in H, as shown in Fig. 12. This hierarchy of bounding spheres can be computed bottom-up when building the ecol hierarchy H at initialization.

Normal cone angle. The cone defined by the semi-angle θ ($t.theta$) about the vertex normal at the endpoint of half-edge $t.h$ bounds the cone of normals [35] of all triangles that are affected by the half-edge collapse t and its descendants in H, as shown in Fig. 13. The bounding normal cones hierarchy is also built bottom-up during the initialization process of H. Note that FastMesh actually only maintains the value of $\sin\theta$ instead of θ itself after initialization since only $\sin\theta$ is needed to compute the view-dependent simplification criterion.

Hierarchical bounding normal cones are considered suboptimal for illumination and view-dependent mesh simplification in [36] because they produce overly conservative measures for surface normal deviation. Furthermore, it is argued that this leads to excessive update rates between successive frames. In contrast, FastMesh uses bounding normal cones very successfully, which can be attributed to the following two reasons: First, the bounding normal cone information can be used very efficiently in four out of the five presented simplification criteria. Second, the use of normal cones allows very fast computation of the error heuristics and

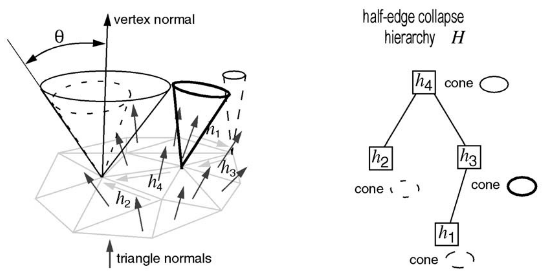


Fig. 13. Bounding normal cone of a sequence of half-edge collapse operations.

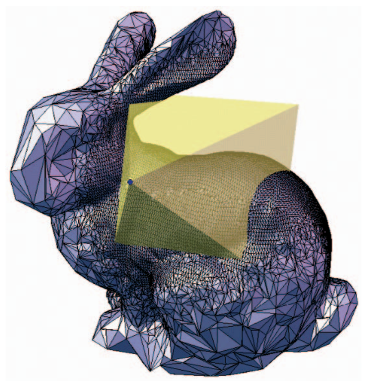


Fig. 14. Example view-frustum simplification with the simulated view frustum shown as transparent yellow pyramid. Rendered 58 percent or 40,881 out of 69,451 faces.

thus sustains a high rate of real-time mesh updates per frame, in contrast to the observations in [36].

For detailed explanations of the following mesh simplification criteria, the reader is referred to Appendix I (which can be found on the Computer Society Digital Library at http://computer.org/tvcg/archives.htm) (see also [8]).

4.1 View Frustum

In order to avoid the bandwidth bottleneck of transferring an unnecessarily large amount of geometry data from main memory to the graphics hardware and to reduce the graphics load of performing view-frustum culling for a large number of invisible triangles, the mesh regions outside of the view frustum can be kept at the coarsest possible resolution. Thus, a half-edge collapse can be performed if its bounding sphere does not intersect the view frustum.

An example view-frustum simplification is shown in Fig. 14. Within the indicated view frustum specified by the transparent yellow pyramid, the triangle mesh is rendered in highest resolution, also including the invisible regions on the back side. The mesh regions outside the view frustum or, more specifically, outside the viewing cone with semi-angle ω are greatly simplified.

4.2 Back-Faces

For large and complex triangle meshes, a large fraction of the triangles that are within the view frustum will be discarded in the graphics rendering pipeline's back-face culling stage. These unnecessary triangles can cost a significant amount of computation time, even if only processed and discarded using back-face culling, since they have to be transmitted to the graphics card, geometrically transformed, and tested for back-face culling. To prevent unnecessary processing of vertices, we can keep back-facing regions of the mesh at the coarsest possible resolution.

Fig. 15 shows an example for back-face simplification. Only the surface regions that are oriented toward the viewpoint are displayed in full resolution, all back-facing regions of the triangle mesh are simplified as much as possible.

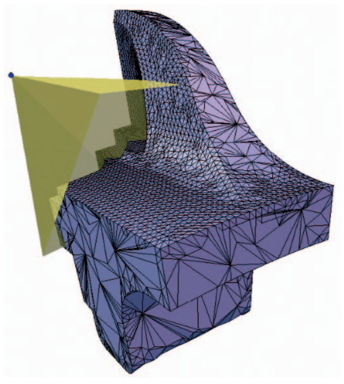


Fig. 15. Example back-face simplification for the displayed view frustum. Rendered 62 percent or 6,750 our of 12,946 faces.

4.3 Screen Projection

If a polygonal mesh is rendered without the use of antialiasing techniques, it is intuitively clear that sufficiently small polygons (i.e., projected area smaller than a pixel) can only create visual artifacts in the rendered image, but not contribute to a smooth display. Even when using antialiasing, it makes little sense to render thousands of insignificantly small triangles with respect to the limited screen display resolution. Furthermore, the performance bottleneck in interactive rendering of large polygonal scenes and objects can also effectively be reduced by simplification of very small triangles. This is particularly true for graphics subsystems that are mainly geometry limited and not pixel fill-rate limited, which is the case for most current systems. Therefore, to improve rendering performance, mesh simplification can be used to remove triangles whose projected area on screen is sufficiently small with respect to an application specific or user given threshold τ (fraction of area on screen). Note that an ecol operation of a node $t \in H$ affects all triangles incident on the removed vertex.

Fig. 16 shows an example with projected screen area error tolerance of $\tau = 0.001 = 1/2^{10}$, measured as a fraction (percentage) of the viewport size—the red square of size τ on the imaginary view plane at the base of the transparently rendered view-frustum pyramid. Only screen projection simplification is turned on in Fig. 16. The example shows the variance in simplification based on the distance from the viewpoint and compares the actual view of the simplified model with the full resolution model.

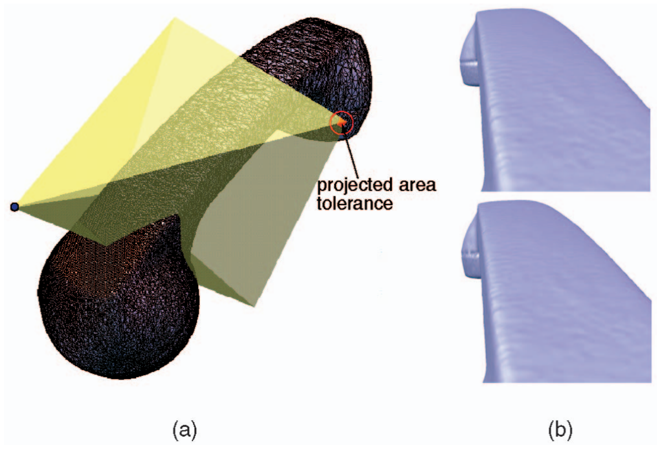


Fig. 16. (a) Example of screen projection based simplification with projection tolerance $\tau = 1/2^{10}$, rendered 32 percent or 53,433 out of 165,963. (b) Top image without simplification (165,963 faces) compared to view with screen projection simplification (53,433 faces).

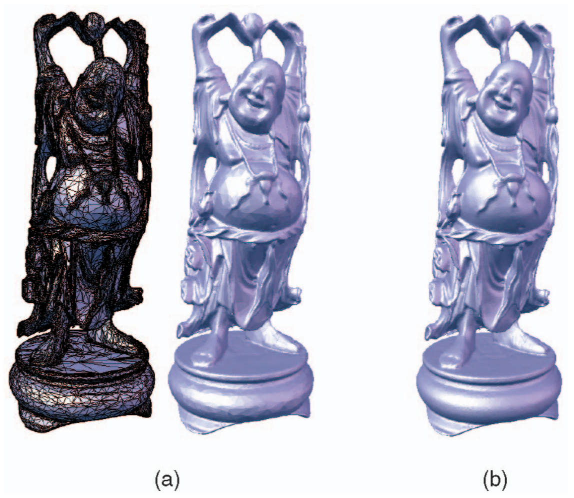


Fig. 17. Example of simplification based on shading tolerance. (a) Tolerated angular normal direction deviation is $\varphi = 15°$, rendered 66 percent or 66,024 out of 100,000 faces. (b) The corresponding full resolution mesh.

4.4 Shading

View-dependent geometric simplification criteria such as screen projection as outlined above may not simplify largely flat surface regions sufficiently because of the vertex position-based error heuristic. Even though triangles may be large and removal of vertices may introduce an intolerable geometric distortion, a triangle mesh might still be simplified if the difference in lighting and shading effects is not visually significant. We can measure the potential deviation in diffuse shading for a particular half-edge collapse operation by the variance in surface normals of the affected triangles.

An example of this diffuse illumination simplification criterion is given in Fig. 17. It shows a model simplified with a tolerance of $\varphi = 15°$ in wire frame and smoothly shaded compared to the full resolution model.

While this shading heuristic as presented above is not dependent on the actual viewpoint, it is a simplification criterion that affects the shading accuracy of the triangulated surface. We argue in Appendix I (which can be found on the Computer Society Digital Library at http://computer.org/tvcg/archives.htm) that such a shading-based mesh simplification heuristic cannot be used to trim the multiresolution hierarchy in a preprocess.

4.5 Silhouettes

The silhouette outline of an object carries a lot of visual information on the object's 3D shape and is perceptually very important. Distortion along the silhouette has a low visual tolerance and can quickly lead to a limited spatial understanding of the object's 3D shape. Therefore, view-dependent mesh simplification should take care of preserving the silhouettes as much as possible.

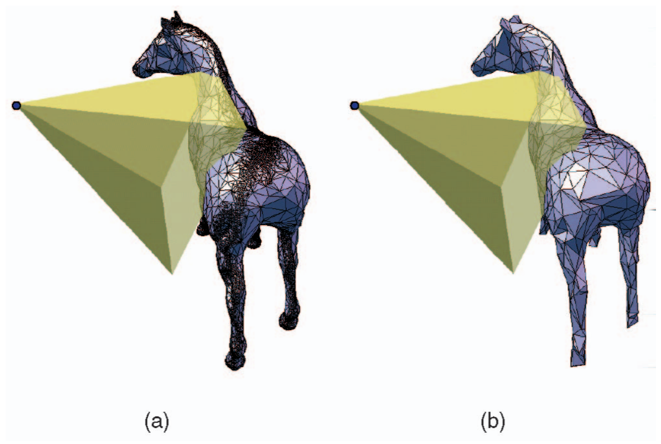


Fig. 18. (a) Example of silhouette preservation, rendered 17 percent or 17,416 out of 96,966 faces. (b) Without silhouette preservation, rendered 1 percent or 1,312 out of 96,966 faces.

The example in Fig. 18a shows strict silhouette preservation. While the triangle mesh has a high resolution along the silhouette area for the given view frustum, other regions are simplified to the coarsest possible resolution.

5 Implementation Details

5.1 Preprocess

The preprocessing stage of the presented view-dependent meshing framework consists of the following steps:

1. Reading the triangle mesh input file, and creating the half-edge data structure.
2. Selecting the partially ordered set of half-edge collapses for simplifying the input mesh M^n to the base mesh M^0.
3. Generating the binary half-edge collapse hierarchy H and computing the bounding sphere radius as well as bounding normal cone semi-angle for each node $t \in H$.

Step 1 is not further explained here; it consists of reading the triangle mesh file, generating three CCW-oriented half-edges for each triangle, and setting up the reverse information of half-edges.

For the mesh simplification process in Step 2, a simplification method based on a static object-space geometric error metric has to be used since view-dependent criteria cannot be used for this view-independent preprocess. This does not seem to be a particular drawback since less important features in object-space are also likely to be of less visual importance in image-space. We use the modified quadric error metric described in Section 2.1 to guide the preprocess mesh simplification phase. Furthermore, this process should create a set of simplification operations that exhibit the least possible partial ordering.

Any partial ordering of edge-collapse operations is expressed by parent-child relations in the binary hierarchy H. The worst case is a binary hierarchy that is degenerated to a linear list, in which case a complete order is defined over all edge-collapse operations and the refinements or simplifications can only be performed in that view-independent order and not view-dependently. The best case is a fully balanced binary hierarchy which exhibits least possible partial ordering among edge-collapse operations. A set of independent half-edge collapses forms a set of nodes in the binary hierarchy that do not have direct parent-child relations and thus do not exhibit any partial ordering in between these operations. Therefore, to reduce dependencies due to partial ordering of nodes $t \in H$, the preprocess selection of simplification operations is performed in batches of largest independent sets of half-edge collapses [14], [15]. In fact, since it produces excellent simplification batches, we use exactly the same batch-wise simplification as in [15] during the preprocess of FastMesh.

At first sight there seems to be a trade off between hierarchy balance and approximation quality and that a greedy mesh simplification would provide better quality LOD meshes. This is not necessarily true since a view-dependent simplification process must prioritize localized mesh refinements over a global refinement order to create high quality view-dependent meshes. Nevertheless, we want to point out here that other (greedy) mesh simplification approaches based on half-edge collapse operations such as [12] and [13] are viable alternatives to select a partially ordered set of edge collapse operations in the preprocess.

Generating the binary half-edge collapse hierarchy H in Step 3 involves setting the correct parent-child relations between the selected half-edge collapse operations. To achieve our storage efficiency, we exploit implicit relations between different data elements based on a specific ordering of the data in arrays. Section 5.2 goes into more detail on ordering and arranging our data structures efficiently. Furthermore, the view-dependent simplification parameters, the bounding sphere *radius*, and the normal cone semi-angle θ (respectively, $\sin\theta$) have to be computed. This is performed by a recursive depth-first traversal of H. At every node $t \in H$, the coefficients $t.radius$ and $t.q$ are initialized according to the vertices and triangles adjacent to the start point $h.v$ of the collapsed edge h and maximized over $t.r.radius$ and $t.l.radius$ for the bounding sphere, respectively, $t.r.q$ and $t.l.q$ for the bounding normal cone. Finally, all entries $t.q$ are converted to $\sin(t.\theta)$.

5.2 Data Structures

The data structures used in FastMesh to maintain the view-dependent multiresolution mesh are fairly simple and reflect the simplicity and efficiency of our approach. In our framework, a view-dependent multiresolution mesh consists of three different data sets: the vertices, the half-edges, and the nodes of the half-edge collapse hierarchy. The n vertices are stored in multiple arrays, one for each attribute such as spatial coordinates, surface normal orientation, color, or texture coordinates. As shown in Fig. 19, the half-edges are stored in an array *hedges* of length $ne = 6n$. The half-edge collapse hierarchy H of size $nm \leq n$ is represented by three different arrays, as shown in Fig. 19. The array *merge* stores the bounding sphere and normal cone attributes as well as the pointer to the parent for each node. Leaf nodes by definition have no child nodes, thus, to save storage space, left and right child pointers are only stored for nonleaf nodes in the two arrays *lchild* and *rchild*.

```
struct halfedge {      // half-edge data structure
   int rev;
   int vtx;
};
struct bintree {       // binary half-edge collapse hierarchy
   bintree *p;         // parent links
   float radius;       // bounding sphere radius
   float sintheta;     // sinus of normal cone semi-angle
};
struct list {          // doubly linked list of active nodes
   list *next, *prev;
   bintree *node;      // pointer to active node in hierarchy
};

int ne;                // number of half-edges
int init;              // position of initial faces in edge array
int nm;                // number of merge-tree nodes
int nl;                // number of merge-tree leaf nodes
int nr;                // number of merge-tree root nodes

halfedge hedges[ne]; // half-edge data structure
bintree merge[nm];     // merge-tree nodes
bintree *lchild[nm-nl]; // child pointres for ..
bintree *rchild[nm-nl]; // .. non-leaf nodes
```

Fig. 19. FastMesh main data structures.

Additionally, at runtime, a doubly linked list maintains the active nodes in the front F.

During the preprocess, the arrays storing half-edges and nodes $t \in H$ are arranged in a particular way, as shown in Fig. 20, to exploit implicit indexing relations between them. The merge-tree nodes $t \in H$ are ordered in such a way that the root nodes reside at the beginning, followed by nonleaf nodes, and with the leaf nodes at the end of the array. The list of root nodes is required by the rendering algorithm to draw the activated faces of the current mesh. Likewise, the elements of the *lchild* and *rchild* arrays are aligned with the first $nm - nl$ elements of the *merge* array. All nodes with index $i \geq nm - nl$ are assumed to have $lchild[i]$ and $rchild[i]$ to be NULL pointers. Additionally, the half-edges are ordered such that, for a given node $merge[i]$, its collapsed half-edge is $heges[6 \cdot i]$ and the reverse of this half-edge is $heges[6 \cdot i].rev = heges[6 \cdot i + 3]$. This implicit relationship between half-edges and merge-tree nodes removes the need to store explicit pointers between the two structures, saving significant storage space compared to [8]. At the same time, this arrangement removes the need to store separate information per face, as was the case in [8]. Furthermore, the initial faces corresponding to the coarsest mesh are placed at the end of the *hedges* array, starting at index location $init$. These faces are never removed from the triangle mesh by any refinement operation, only their connectivity information is modified.

For each node $t \in H$, FastMesh needs to store a flag indicating whether that node is currently split or collapsed. This Boolean value can be stored in a single spare bit. That spare bit is conveniently located in the sign bit of the node's bounding sphere radius. We assume that the bounding sphere radius cannot be negative. Therefore, FastMesh can test the sign of the radius to determine the state of a node and flip the sign when that state, split or collapsed, changes at runtime.

To be able to propagate forced splits as described in Section 3.4, it must be known which faces are removed or inserted by which node. This relation is simply given by the fact that node i collapses faces $2 \cdot i$ and $2 \cdot i + 1$. Thus, given

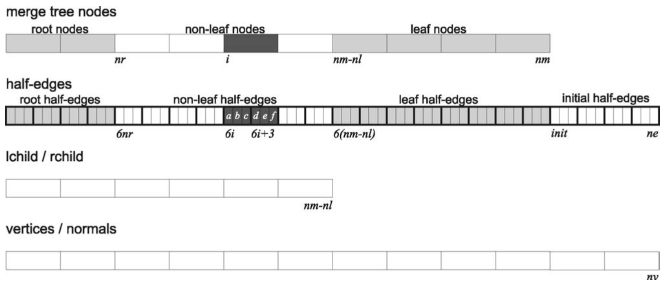


Fig. 20. FastsMesh data structures are stored in three parallel arrays, one each for merge tree nodes, half-edges, and vertices. Within the half-edge array, each set of three consecutive half-edges comprises one triangle face.

a half-edge a, it can be tested if the corresponding face $f = a$ DIV 3 is currently active by testing if it is an initial face, $f \geq init$ or if the corresponding node $i = f$ DIV 2 is not collapsed. If the node is collapsed, then a forced split is propagated to this node.

5.3 Rendering

In the main rendering loop, the front of active nodes F has to be traversed and tested for each new frame with changed viewing parameters. After traversing the active nodes, the front F has to be adjusted within the hierarchy H. Algorithm 1 in Appendix II (which can be found on the Computer Society Digital Library at http://computer.org/tvcg/archives.htm) shows the steps performed on each active node of the traversed front F. For each node, the simplification heuristics described in Section 4 are evaluated. If a node represents a collapsed half-edge, it is recursively refined if required by the simplification heuristic. Otherwise, the half-edge is collapsed if allowed or its children are tested to be refined recursively.

The procedure to evaluate the view-dependent simplification heuristics is given in Algorithm 2 in Appendix II (which can be found on the Computer Society Digital Library at http://computer.org/tvcg/archives.htm). The code has been modified from standard C++ to be more concise; it uses vector variables (i.e., $\bar{v}$) and vector operations (+, -, and dot product $\cdot$), and self-explanatory variable names such as $\cos\gamma$ or $\sin\theta^2$ where appropriate. Additionally, it assumes that the view parameters are given by the viewpoint ($\overline{eye}$), the view direction ($\overline{dir}$), view frustum aperture semi-angle ω (respectively, $\tan\omega$ and $\cos\omega$), and focal length d.

At any time, the simplified mesh consists of two types of triangles: 1) the initial triangles which are present in the coarsest mesh M^0 and 2) any detail triangles which correspond to faces introduced by vertex split operations. The initial triangles are stored consecutively in memory, each as a sequence of three half-edges, pointed to by the index $init$, as also shown in Fig. 20. These faces are rendered every frame by iterating through the array *hedges* starting at index $init$. Detail triangles are rendered using a depth-first traversal of the half-edge collapse hierarchy H. Because each node $t \in H$ corresponds to a vertex split which introduces two faces, all nodes that have been split exactly indicate the active faces that represent the current mesh (in addition to the initial faces of M^0). A noncollapsed node with index i corresponds to two active triangles

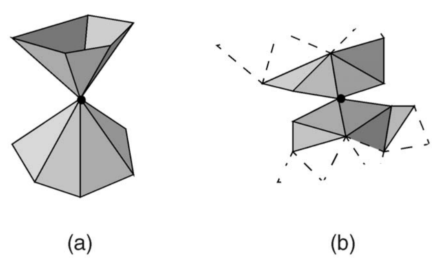


Fig. 21. Nonmanifold vertices: (a) shared between different surface parts or (b) as part of multiple boundaries or surfaces.

referenced by half-edges $6 \cdot i$ and $6 \cdot i + 3$. As shown in Algorithm 3 in Appendix II (which can be found on the Computer Society Digital Library at http://computer.org/tvcg/archives.htm), the recursive traversal stops if a collapsed node is found.

5.4 Nonmanifold Meshes

As presented so far, the view-dependent simplification method using a half-edge-based triangle mesh data structure and a binary half-edge collapse hierarchy works well on manifold triangular meshes of arbitrary topology (genus). Boundary edges can also be handled consistently in the framework with little additional effort as outlined in Section 3.2. In this section, we want to provide additional information on how nonmanifold vertices are handled and suggest a pragmatic solution for nonmanifold edges.

First, let us consider singular, nonmanifold vertices only, as shown in Fig. 21. Around such singular vertices, the surface appears to have multiple, individually manifold, components that touch each other exactly at one vertex. Accordingly, the mesh connectivity with respect to traversal across edges is consistent, each edge is shared by exactly two triangles, or only one if the edge is on the boundary. In fact, since edges refer to vertices by indices and edge collapse operations do not alter any vertex coordinates, the vertex is treated as if it where duplicated for the different mesh components. Therefore, our approach gracefully handles this situation as if the mesh indeed consisted of different manifold subparts that actually touch each other at this point in space. No special care has to be taken to cope with such isolated nonmanifold vertices.

Nonmanifold edges shared by more than two triangle faces are difficult to handle for many if not most algorithms and data structures for triangle meshes and constitute a problem in particular for mesh simplification methods. Nonorientable edges are also a problem for the half-edge data structure and will be considered nonmanifold edges. The problems with nonmanifold edges in FastMesh are mostly due to the connectivity relations stored in the reverse half-edge entries. Nonmanifold edge connectivity cannot be represented with a basic half-edge data structure and thus prevents a correct traversal of mesh elements across edges. For practical purposes, we suggest preprocessing the input triangle mesh and converting it to manifold triangle mesh components, as shown in [37]. Since the number of nonmanifold edges is typically small compared to the size of the entire mesh, this approach can reasonably be used in practice. Note that this approach has, in fact, successfully been used for compression of nonmanifold triangle meshes, as shown in [38]. For a sufficiently small number of nonmanifold edges, the reverse information could be set to some special value when generating the half-edge data structure during the preprocess. Similarly to boundary edges, the reverse of nonmanifold edges could be set to an invalid value. This way, nonmanifold edges would be treated the same as boundary edges in the presented FastMesh framework.

For handling nonmanifold edges without preprocessing the triangle mesh or converting them to boundary edges, the following operations must be supported efficiently:

- visiting all edges or faces incident on a vertex (i.e., for vertex reassignment after edge collapse or error calculation),
- collapsing all faces incident to a collapsed edge and the inverse vertex split operation,
- updating mesh connectivity information after simplification and refinement operations.

This requires a completely different triangle mesh representation, such as the radial edge [39] or quarter edge [40] structure. Since this paper focuses on an optimized implementation for manifold meshes, we leave this for future work.

6 Experimental Results

We tested our view-dependent meshing approach on various models of different sizes and varying shapes. The experiments include measurements of the main memory and disk space requirements of the FastMesh data structures, as well as runtime performance of the view-dependent error calculations and timing of the dynamic mesh updates using the half-edge collapse hierarchy.

Fig. 22 and Fig. 23 show examples of the dragon model and a triangulated terrain surface simplified using the proposed view-dependent mesh simplification and rendering method. In these high-resolution examples, strict silhouette preservation is relaxed by coupling it with a threshold on the length of the collapsed edge. If the squared length of the half-edge collapse is smaller than $\tau/100$, then silhouette preservation is not performed.

6.1 Space Cost

The main memory usage of FastMesh is determined by the number of mesh elements and the size of the data structures of Fig. 19. Note that the number of nodes in the half-edge collapse hierarchy is limited by the number of vertices. For a mesh with n vertices, about $n/2$ leaf as well as $n/2$ nonleaf nodes, about $2n$ triangles and $6n$ half edges, the runtime space cost consists of:

- $24n$ bytes for vertex coordinates and normals,
- $8 \cdot 6n$ bytes for the half-edges table,
- $12 \cdot n/2$ bytes for the leaf nodes,
- $20 \cdot n/2$ bytes for the nonleaf nodes,

Thus, the expected overall main memory size is only $88n$ bytes. Table 1 shows from 87 up to 89 bytes per vertex in our experiments, which is due to the fact that the number nl of leaf nodes is less than $n/2$, but larger than $n/3$, in practice. This is an improvement of about 20 percent over

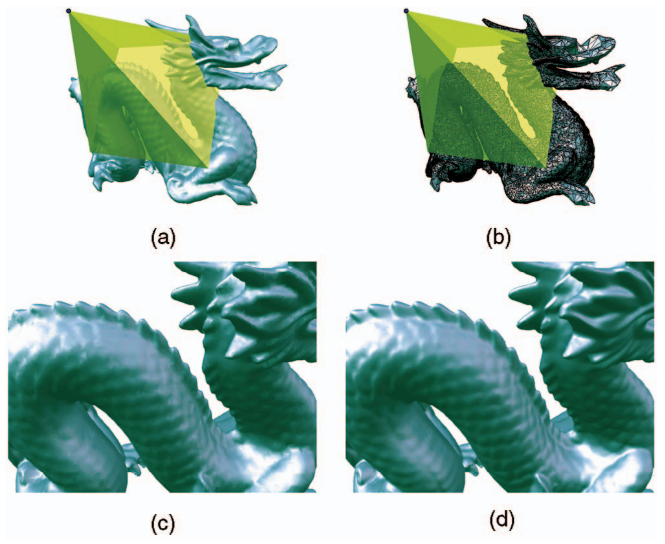


Fig. 22. Dragon model example rendered with all visual error metrics enables, screen projection tolerance $\tau = 1/2^{10}$, and angular deviation tolerance $\varphi = 0.1^\circ$. Images (a) and (b) shown the dragon renderd with 139,082 faces out of 871,414 (15 percent). Images (c) and (d) show renderings from the actual viewpoint in (c) using the same tolerances with 139,082 faces (15 percent) and, in (d) at full resolution of front-facing surface within view frustum with 419,180 faces (48 percent). Strict silhouette preservation is relaxed by coupling it with the squared length of the silhouette edge as projected on the view-plane and thresholding it with 0.01τ.

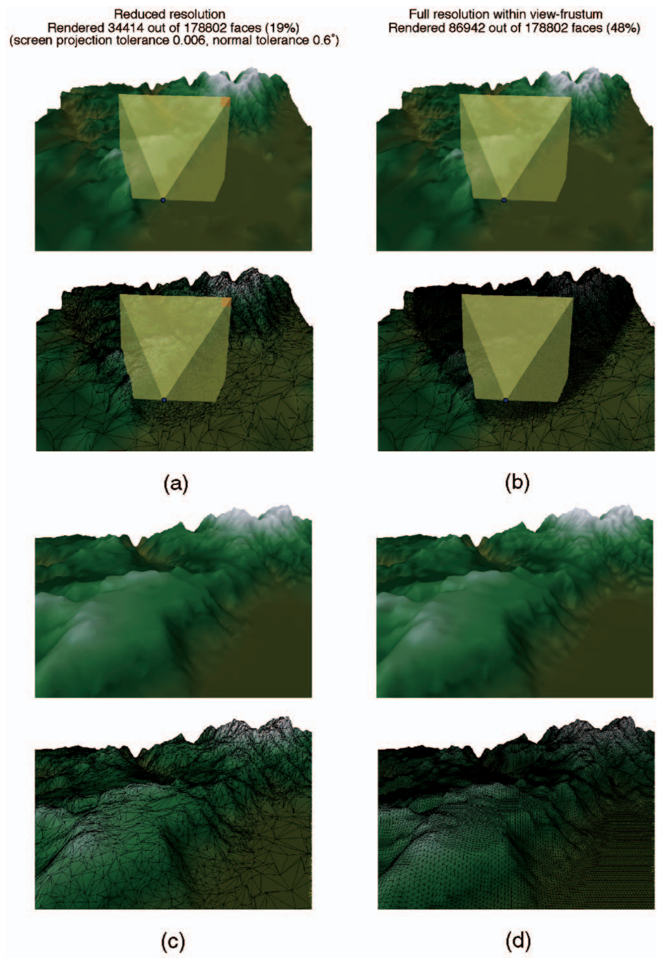

Reduced resolution
Rendered 34414 out of 178802 faces (19%)
(screen projection tolerance 0.006, normal tolerance 0.6°)
Full resolution within view-frustum
Rendered 86942 out of 178802 faces (48%)
(a)
(b)
(c)
(d)

Fig. 23. Images (a) and (b) show the view-frustum and mesh simplification from a bird's eye view and images in (c) and (d) show the terrain as viewed from the actual viewpoint. The right-hand column images (b) and (d) have only view-frustum simplification enabled, thus showing the terrain in full detail within the visible view. The left-hand column has all view-dependent simplifications enabled and strict silhouette preservation is relaxed by coupling it with the square length of the silhouette edge as projected on the view-plane and thresholding it with 0.01τ.

the initial FastMesh representation given in [8]. Additionally, the number of list entries used to define the active front F depends on the number m of rendered vertices. The rendered vertices are defined by a subhierarchy of $\leq m$ nodes that have been split and, given that the list has one entry for each leaf node in this binary subhierarchy, the cost of the active front list for m rendered vertices is approximately $12/2m = 6m$ bytes.

The main memory cost of FastMesh is much smaller than the approach presented in [19] which requires $216n$ bytes.[2] Furthermore, despite the significantly reduced memory cost, our approach calculates more visual error heuristics than [19] at runtime. In [20], an improved method was presented that requires $88n$ bytes for the static part of the representation and, additionally, $100m$ bytes[3] for the dynamically changing triangle mesh with m vertices. Note that this space reduction is achieved at the expense of continuous memory allocation and deallocation of mesh data structures at runtime. Our representation does not have to dynamically allocate or deallocate any data structures representing the active triangle mesh or the binary half-edge collapse hierarchy, it only updates the front of active nodes dynamically and achieves a much better runtime space cost with only $6m$ bytes instead of $100m$ bytes in addition to the $88n$ bytes of static memory cost.

We can also compare the space cost of FastMesh with the results provided in [18] for the View-dependence Tree (VDT) [18] and Merge Tree [16], [17] approaches. The VDT data structure requires about $90n$ bytes only for the nodes of the vertex hierarchy. The overall size of the VDT data structure is about $138n$ bytes or more ($24n$ bytes for vertex coordinates and normals, $24n$ for the indexed triangles, $90n$ bytes for the VDT, plus a variable number of bytes for the active element lists). The Merge Tree [16], [17] approach requires much more storage since its data structure is about two times as large as a VDT representation. The Multi-Triangulation (MT) approach presented in [23] achieves a compact representation size of about $75n$ bytes for a test mesh of 35,162 vertices for the proposed implicit graph data structure or $194n$ bytes, respectively, for the explicit graph data structure. Note that, for the implicit MT representation, this space cost does not include the storage of active, rendered triangle faces at runtime, which has to be allocated dynamically.

Static storage of a FastMesh representation on disk as shown in Table 1 requires even less space. Note that the reverse fields of half-edges can be recovered at initialization and need not to be stored in a file on disk. The binary hierarchy is stored as a sequence of half-edge collapse indices and bounding sphere and normal cone parameters.

2. Without material id.
3. Without morphing and texture information.

TABLE 1
Number of Vertices, Faces, and Half-Edges Given for Each Model

Model	Full resolution			FastMesh			
	vertices	faces	h-edges	nodes	memory	bytes/v	disk
fandisk	6475	12946	38838	6469	0.55	89	0.4
bunny	35947	69451	208353	34565	2.97	87	2.0
horse	48485	96966	290898	48470	4.11	89	2.8
happy1	49794	100000	300000	49122	4.22	89	2.0
phone	83044	165963	497889	82883	7.04	89	4.8
terrain	90000	178802	536406	88652	7.58	88	2.5
dragon	437645	871414	2614242	428229	36.9	89	26
happy2	543652	1087716	3263148	542735	46	89	31

Number of nodes in the half-edge collapse hierarchy as well memory cost of the FastMesh data structure, in main memory, and on disk given in MBytes.

6.2 Time Cost

The runtime performance tests were performed on a Sun Ultra60 workstation, equipped with a 450MHz Ultra-SPARC-II CPU, an Expert3D PCI-bus graphics card. The CPU time usage was measured with the high-resolution timing function gethrtime() available on Sun/Solaris machines. Where possible, comparisons with previously published results are indicated.

In the three left-hand columns of Table 2, we report construction times of a FastMesh data structure when initialized from a plain text triangle mesh file (columns *ecol selection* and *hierarchy*) as well as initialization from a binary FastMesh representation stored on disk (column *from disk*). Initialization from a plain triangle mesh file includes the selection of half-edge collapses as outlined in Section 2.1, as well as the construction of the half-edge collapse hierarchy, and computation of simplification parameters as described in the introduction of Section 4. Initialization from a binary FastMesh file on disk consists of reading and reconstructing the half-edge data structure, as well as reading the simplification parameters and reconstructing the binary half-edge hierarchy from the sequence of half-edge collapses. The improved preprocessing time over [8] is mainly due to the use of linear-time radix sorting of edge-collapse candidates after each batch of independent simplification operations. These preprocessing results compare favorably with the times reported in [18] for the View-dependence Tree (VDT) approach. FastMesh requires less than 120 seconds to process the *dragon* data set on the Sun Ultra60, while the VDT method needs almost 400 seconds on an SGI Onyx2.

Meshing and rendering runtime performance was also measured in CPU time usage using high-resolution timing. The test consisted of rotating the view frustum around the center of the objects at two different screen projection error thresholds $\tau = 0.0$ and $\tau = 1/2^{10}$. One thousand frames were rendered for each test run. The three tasks that have to be carried out per frame and that have been timed are:

1. Rendering, which includes setting up the graphics context and calls to the OpenGL glVertex() and glNormal() functions. The active triangle mesh was rendered using shaded triangles without texturing.
2. Calculating and testing the view-dependent error heuristic for all active nodes.
3. Updating the mesh using vertex split and half-edge collapse operations according to the results of the view-dependent tests on all active nodes.

For each task, we counted the number of elements that have been processed and measured the CPU time that was used. Table 2 presents the achieved results averaged per frame and shows the runtime cost of each individual task in relation to the overall CPU time usage. As can be seen, most of the CPU time is spent on the actual rendering task, which consumes up to 64 percent. Even though a significant number of view tests have to be performed each frame, over 100,000 each frame for the happy2 model at $\tau = 0.0$, our approach is very efficient and only uses between 15 to

TABLE 2
Initialization Performance Given in Seconds to Preprocess a Given Triangle Mesh and Construct the Half-Edge Hierarchy or Read a FastMesh Data Structure from Disk

Model	Initialization			Run-time (averaged per frame)										
	ecol selection	hierarchy	from disk	τ	rendering				error heuristic			updating mesh		
					FPS	$\lvert\Delta\rvert$	time	% of frame	$\lvert$tests$\rvert$	time	% of frame	$\lvert$updates$\rvert$	time	% of frame
fandisk	1.1s	0.07s	0.1s	0.0	41	4589	11.5 ms	47%	1328	1.2 ms	5%	48	0.1 ms	0.4%
				0.001	34	3967	8.7 ms	30%	1211	1.1 ms	4%	48	0.1 ms	0.3%
bunny	7.3s	0.6s	0.6s	0.0	22	18738	21.2 ms	47%	5251	5.7 ms	13%	265	0.8 ms	1.8%
				0.001	26	17354	19.1 ms	50%	5461	6.0 ms	16%	298	0.9 ms	2.3%
horse	11s	0.8s	1.4s	0.0	13	36509	43.0 ms	56%	9102	11.4 ms	15%	467	1.4 ms	1.8%
				0.001	16	25539	31.0 ms	50%	9752	12.3 ms	20%	637	1.9 ms	3.0%
happy1	12s	0.8s	1.6s	0.0	8.4	56432	66.1 ms	56%	13392	18.8 ms	16%	426	1.3 ms	1.1%
				0.001	8.7	49881	58.7 ms	51%	14376	20.2 ms	18%	512	1.6 ms	1.4%
phone	19s	1.5s	1.7s	0.0	9.8	53213	64.9 ms	64%	12634	17.7 ms	17%	574	2.0 ms	2.0%
				0.001	12	33077	40.7 ms	49%	12755	17.8 ms	21%	784	2.6 ms	3.1%
terrain	19s	1.5s	1.9s	0.0	5.5	78108	108 ms	59%	18149	25.2 ms	14%	718	2.5 ms	1.4%
				0.001	6.9	53272	74.7 ms	52%	17159	24.1 ms	17%	848	3.1 ms	2.1%
dragon	108s	8.8s	9.8s	0.0	0.9	413863	605 ms	54%	92266	171 ms	15%	3945	16 ms	1.4%
				0.001	1.9	134340	193 ms	37%	60065	111 ms	21%	3455	15 ms	2.9%
happy2	133s	11s	12.5s	0.0	0.9	478771	718 ms	64%	115922	205 ms	18%	4528	19 ms	1.7%
				0.001	1.9	152011	241 ms	46%	78457	141 ms	27%	3169	14 ms	2.7%

Runtime performance for rendering, computing error heuristics, and updating the triangle mesh data structure is given by the per frame average number of elements processed (column |x|), the time in milliseconds to perform each task, and its percentage of overall CPU cost.

20 percent of the frame time. In particular, the dynamically changing half-edge-based triangle mesh data structure is extremely efficient; it only consumes 3 percent and less of the per frame CPU cost to update the view-dependent triangulation. One can observe that, with increasing τ, the view tests consume more CPU time relative to the rendering task. However, the view-dependent mesh simplification proves to be very beneficial as the entire per frame time of rendering, testing, and updating with $\tau = 1/2^{10}$ is lower than even just the triangle rendering time with $\tau = 0.0$ for the larger models. On the test hardware, an amortized mesh update, sum of many view tests, and one mesh update are performed in $20\mu s$ to $50\mu s$, while an ecol or vsplit operation by itself only needs about $3\mu s$ to $5\mu s$ on average.

Experiments for the Multi-Triangulation (MT) approach [23] are reported on an SGI Indigo2. For their fastest algorithm, extraction times of about $60\mu s$ per rendered triangle are reported. Amortized per rendered triangle, our approach needs, for mesh extraction, which includes view tests and mesh updates, about $0.5\mu s$ to $1.1\mu s$. Despite the more than two times slower hardware architecture (195Mhz R10000 compared to a 450Mhz UltraSparc), this large performance difference is a clear indication of the complex mesh update operations of the MT approach, even without computing multiple visual error heuristics as done in FastMesh.

The approach in [34] reports $\leq 4,000$ triangle removals per second on a 250Mhz MIPS CPU using a similar half-edge data structure. As reported above, our approach needs $3\mu s$ to $5\mu s$ per mesh update, which inserts or removes two triangles at a time. Thus, we achieve over 400,000 raw triangle removals or insertions per second, on a 450MHz UltraSparc CPU. For the entire update operation, which includes view tests and triangulation, we achieve 40,000 up to 100,000 per second. Note that the best performance reported in [34] was for a representation of the triangle mesh with 68 bytes per triangle or 136 bytes per vertex, while our representation only requires 88 bytes per triangle and not only includes the triangle mesh data but also the view-dependent multiresolution hierarchy data structure.

Experiments in [21] report a performance of $22\mu s$ and $17\mu s$ time cost for vsplit or ecol operations on an 866MHz Pentium III system. Our approach also compares favorably with these results since it only requires $3\mu s$ to $5\mu s$ to perform a vsplit or ecol operation on a much slower clocked 450Mhz UltraSparc Sun workstation.

7 Discussion

In this paper, we present an efficient view-dependent meshing framework and implementation for interactive visualization of highly complex triangle meshes called FastMesh. We introduced a half-edge data structure-based hierarchical multiresolution triangulation framework, developed algorithms for adaptively refining and simplifying the triangle mesh interactively using this half-edge collapse hierarchy, and devised a set of effective view-dependent error heuristics and a highly efficient implementation of it to guide dynamic mesh simplification. Experiments on a variety of meshes have shown the efficiency of the presented view-dependent mesh simplification, as well as the compactness of the required data structures. Compared to previous methods, our approach exhibits fewer dependencies between refinement and simplification operations, which optimizes the adaptivity of the view-dependent mesh generation. For example, collapsing a half-edge does not have to take into account any dependencies with respect to other operations, it only has to pass the standard topological validity test for edge collapse operations. Furthermore, FastMesh uses simpler and more compact data structures than other methods, which results in significantly lower memory requirements than other approaches. Despite the small memory footprint of our approach, FastMesh works extremely well and considerably outperforms other view-dependent meshing and rendering approaches as shown by our experiments.

The presented FastMesh approach has successfully been extended to an out-of-core representation [41] that allows interactive view-dependent rendering of very large meshes stored on external memory and future work in this area includes extension to nontriangular and mixed element meshes, handling of arbitrary nonmanifold polygonal meshes, incorporation of texture and additional illumination-dependent error heuristics, as well as dynamic triangle strip generation [42].

Acknowledgments

The authors are very thankful to the sources for the geometric data models. In particular, they would like to thank the Stanford Computer Graphics Laboratory which maintains freely available models in the Stanford 3D Scanning Repository,[4] Greg Turk for his Large Geometric Models Archive,[5] and the US Geological Survey mapping center[6] for providing high-resolution terrain elevation models.

References

[1] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk, “The Digital Michelangelo Project: 3D Scanning of Large Statues,” *Proc. SIGGRAPH 2000,* pp. 131-144, 2000.

[2] J. Clark, “Hierarchical Geometric Models for Visible Surface Algorithms,” *Comm. ACM,* vol. 19, no. 10, pp. 547-554, 1976.

[3] T. Funkhouser and C. Sequin, “Adaptive Display Algorithm for Interactive Frame Rates during Visualization of Complex Virtual Environments,” *Proc. SIGGRAPH ’93,* pp. 247-254, 1993.

[4] L. De Floriani and E. Puppo, “Hierarchical Triangulation for Multiresolution Surface Description,” *ACM Trans. Graphics,* vol. 14, no. 4, pp. 363-411, 1995.

[5] P.S. Heckbert and M. Garland, “Survey of Polygonal Surface Simplification Algorithms,” *SIGGRAPH ’97 Course Notes 25,* 1997.

[6] P. Cignoni, C. Montani, and R. Scopigno, “A Comparison of Mesh Simplification Algorithms,” *Computers & Graphics,* vol. 22, no. 1, pp. 37-54, 1998.

[7] P. Lindstrom and G. Turk, “Evaluation of Memoryless Simplification,” *IEEE Trans. Visualization and Computer Graphics,* vol. 5, no. 2, pp. 98-115, Apr.-June 1999.

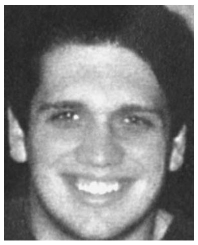
[8] R. Pajarola, “Fastmesh: Efficient View-Dependent Meshing,” *Proc. Pacific Graphics 2001,* pp. 22-30, 2001.

[9] D.P. Luebke, “A Developer’s Survey of Polygonal Simplification Algorithms,” *IEEE Computer Graphics and Applications,* vol. 21, no. 3, pp. 24-35, May/June 2001.

4. http://www-graphics.stanford.edu/data/3Dscanrep/
5. http://www.cc.gatech.edu/projects/
6. http://mapping.usgs.gov/

[10] H. Hoppe, "Progressive Meshes," *Proc. SIGGRAPH '96,* pp. 99-108, 1996.
[11] M. Garland and P.S. Heckbert, "Surface Simplification Using Quadric Error Metrics," *Proc. SIGGRAPH '97,* pp. 209-216, 1997.
[12] L. Kobbelt, S. Campagna, and H.-P. Seidel, "A General Framework for Mesh Decimation," *Proc. Graphics Interface '98,* pp. 43-50, 1998.
[13] W. Dong, J. Li, and J. Kuo, "Fast Mesh Simplification for Progressive Transmission," *Proc. Int'l Conf. Multimedia and Expo (ICME 2000),* 2000.
[14] R. Pajarola and J. Rossignac, "Compressed Progressive Meshes," *IEEE Trans. Visualization and Computer Graphics,* vol. 6, no. 1, pp. 79-93, Jan.-Mar. 2000, corrections printed vol. 6, no. 2, pp. 190-192, Apr.-June 2000.
[15] R. Pajarola and J. Rossignac, "Squeeze: Fast and Progressive Decompression of Triangle Meshes," *Proc. Computer Graphics Int'l (CGI 2000),* pp. 173-182, 2000.
[16] J.C. Xia and A. Varshney, "Dynamic View-Dependent Simplification for Polygonal Models," *Proc. IEEE Visualization '96,* pp. 327-334, 1996.
[17] J.C. Xia, J. El-Sana, and A. Varshney, "Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models," *IEEE Trans. Visualization and Computer Graphics,* vol. 3, no. 2, pp. 171-183, Apr.-June 1997.
[18] J. El-Sana and A. Varshney, "Generalized View-Dependent Simplification," *Proc. EUROGRAPHICS '99,* pp. 83-94, 1999.
[19] H. Hoppe, "View-Dependent Refinement of Progressive Meshes," *Proc. SIGGRAPH '97,* pp. 189-198, 1997.
[20] H. Hoppe, "Efficient Implementation of Progressive Meshes," *Computers & Graphics,* vol. 22, no. 2, pp. 27-36, 1998.
[21] J. Kim and S. Lee, "Truly Selective Refinement of Progressive Meshes," *Proc. Graphics Interface 2001,* pp. 101-110, 2001.
[22] L. De Floriani, P. Magillo, and E. Puppo, "Building and Traversing a Surface at Variable Resolution," *Proc. IEEE Visualization '97,* pp. 103-110, 1997.
[23] L. De Floriani, P. Magillo, and E. Puppo, "Efficient Implementation of Multi-Triangulations," *Proc. IEEE Visualization '98,* pp. 43-50, 1998.
[24] L. De Floriani, P. Magillo, F. Morando, and E. Puppo, "Dynamic View-Dependent Multiresolution on a Client-Server Architecture," *Computer-Aided Design,* vol. 32, no. 13, pp. 805-823, 2000.
[25] D. Luebke and C. Erikson, "View-Dependent Simplification of Arbitrary Polygonal Environments," *Proc. SIGGRAPH '97,* pp. 199-208, 1997.
[26] R. Szeliski and H.-Y. Shum, "Creating Full View Panoramic Image Mosaics and Environment Maps," *Proc. SIGGRAPH '97,* pp. 251-258, 1997.
[27] 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," *Proc. SIGGRAPH '96,* pp. 109-118, 1996.
[28] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineev-Weinstein, "Roaming Terrain: Real-Time Optimally Adapting Meshes," *Proc. IEEE Visualization '97,* pp. 81-88, 1997.
[29] P. Lindstrom and V. Pascucci, "Visualization of Large Terrains Made Easy," *Proc. IEEE Visualization 2001,* pp. 363-370, 2001.
[30] R. Klein, D. Cohen-Or, and T. Hüttner, "Incremental View-Dependent Multiresolution Triangulation of Terrain," *Proc. Pacific Graphics '97,* pp. 127-136, 1997.
[31] K. Weiler, "Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments," *IEEE Computer Graphics and Applications,* vol. 5, no. 1, pp. 21-40, Jan. 1985.
[32] D.E. Muller and F.P. Preparata, "Finding the Intersection of Two Convex Polyhedra," *Theoretical Computer Science,* vol. 7, no. 2, pp. 217-236, 1978.
[33] B. Baumgart, "A Polyhedron Representation for Computer Vision," *Proc. Am. Federation Information Processing Socs. (AFIPS),* vol. 44, pp. 589-596, 1972.
[34] S. Campagna, L. Kobbelt, and H.-P. Seidel, "Directed Edges—A Scalable Reprensentation for Triangle Meshes," *J. Graphics Tools,* vol. 3, no. 4, pp. 1-12, 1998.
[35] L.A. Shirman and S.S. Abi-Ezzi, "The Cone of Normals Technique for Fast Processing of Curved Patches," *Proc. EUROGRAPHICS '93,* pp. 261-272, 1993.
[36] R. Klein, A. Schilling, and W. Strasser, "Illumination Dependent Refinement of Multiresolution Meshes," *Proc. Computer Graphics Int'l (CGI '98),* pp. 680-687, 1998.
[37] A. Guéziec, G. Taubin, F. Lazarus, and W. Horn, "Converting Sets of Polygons to Manifold Surfaces by Cutting and Stitching," *Proc. IEEE Visualization '98,* pp. 383-390, 1998.
[38] A. Guéziec, F. Bossen, G. Taubin, and C. Silva, "Efficient Compression of Non-Manifold Polygonal Meshes," *Proc. IEEE Visualization '99,* pp. 73-80, 1999.
[39] K. Weiler, "The Radial Edge Structure: A Topological Representation for Non-Manifold Geometric Boundary Modeling," *Geometric Modeling for CAD Applications,* pp. 3-36, 1988.
[40] M. Niizeki, F. Konishi, M. Yoshida, and F. Yamaguchi, "The Quarter-Edge Data Structure: The Principle of Duality in Data Structures and Euler Operators of Solid Modelers," *J. Advanced Automation Technology,* vol. 6, no. 4, pp. 223-228, 1994.
[41] C. DeCoro and R. Pajarola, "XFastMesh: Fast Ciew-Dependent Meshing from External Memory," *Proc. IEEE Visualization 2002,* pp. 363-370, 2002.
[42] M. Shafae and R. Pajarola, "DStrips: Dynamic Triangle Strips for Real-Time Mesh Simplification and Rendering," *Proc. Pacific Graphics 2003,* pp. 271-280, 2003.

Renato Pajarola received the Dr.sc.techn. degree in computer science in 1998 from the Swiss Federal Institute of Technology (ETH) Zurich. Following graduation, he was a post-doctoral researcher and part-time faculty member at the Graphics, Visualization Usability Center at the Georgia Institute of Technology for a year. Since 1999, he has been an assistant professor in information and computer science at the University of California, Irvine. His current research interests include real-time 3D graphics, multiresolution modeling, image-based rendering, image and geometry compression, interactive remote visualization, large-scale terrain and volume visualization, as well as interactive 3D multimedia. He has published a wide range of technical papers in top journals and conferences. He has served as a reviewer and program or organization committee member for several international conferences and is a frequent referee for journal articles. He is a member of the IEEE Computer Society, ACM, and the ACM Special Interest Group on Computer Graphics.



Christopher DeCoro graduated summa cum laude from the University of California, Irvine with a bachelor of science degree in computer science (2002). He is currently pursuing graduate studies at the Courant Institute of Mathematical Sciences at New York University. He is a student member of the IEEE and ACM.

Appendix I
View-dependent Error Heuristics

This appendix explains in detail the computation of the view-dependent and visual error heuristics outlined in Section refsec:error.

A. View frustum

The distance to the view frustum can easily be computed given its four bounding planes as presented in [19]. However, the 16 plane parameters are usually not given in a 3D graphics system. We present an alternative approach where we bound the view frustum by a cone with semi-angle ω about the viewing direction n. The viewpoint e, normalized view-direction n, and field-of-view (FOV) angle 2ω are the standard viewing parameters defining a perspective projection in most graphics systems.

As shown in Figure 24, a half-edge collapse with endpoint v and bounding sphere radius r is outside the view frustum and can be performed if $\gamma - \varphi > \omega$ or $\cos(\gamma - \varphi) < \cos(\omega)$. Applying a few trigonometric rules such as $\cos(\alpha) = \sin(90 - \alpha)$ and $\sin(\alpha + \beta) \leq \sin(\alpha) + \sin(\beta)$ this can be rewritten to

$$\cos\gamma + \sin\varphi < \cos\omega. \tag{3}$$

Equation 3 can efficiently be evaluated without trigonometric functions by using one dot-product and two divisions: $\cos(\gamma) = ((v - e) \cdot n)/|v - e|$, $\sin(\varphi) = r/|v - e|$ and $\cos(\omega)$ can be precomputed once as long as the FOV aperture angle does not change between frames. A few additional relations such as $\cos(\gamma) > \cos(\omega) \Rightarrow$ *split node*, $|v - e| < r \Rightarrow$ *split node*, and $(v - e) \cdot n < -r \Rightarrow$ *collapse node* need to be tested before Equation 3 can be used to decide if a node can be collapsed.

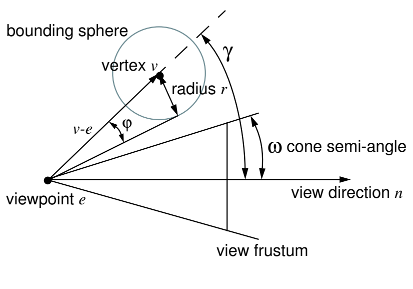


Fig. 24. Outside view-frustum simplification.

Equation 3 is conservative for $\gamma \leq 90$ and $\gamma - \varphi \geq \omega$ in that it approximates the value $\cos(\gamma - \varphi)$ by $\cos(\gamma) + \sin(\varphi)$. The worst case is achieved if $\gamma - \varphi = \omega$, which for a typical FOV semi-angle $\omega = 30$ results in an overestimation of the actual value $\cos(\gamma - \varphi)$ by at most 13%. For $\gamma - \varphi < \omega$ or $\varphi \geq \gamma$, and since $\varphi \leq 90$ and $\omega \leq 90$, the conservative overestimation does not affect the result, and Equation 3 provides the correct inequality relation that prevents simplification operations on elements within the view frustum.

For the special configuration of $\gamma > 90$ and $-r < (v - e) \cdot n < 0$ we can approximate the test by $|v - e| > r/\cos(\omega)$.

B. Back-faces

To keep back-facing regions of the mesh at the coarsest possible resolution, a node t and half-edge $t.h$ with endpoint $t.h.v$ can be collapsed if its entire associated normal cone is back-facing, that is if no normal within the cone can be front-facing. Or in other words, none of the triangles that were removed or distorted by a simplification operation corresponding to a node $s \in H_t$ is front-facing. As can be seen from Figure 25, given the normal cone with semi-angle θ and the angle γ between the normalized vertex normal n_v and the vector $\bar{ev}$ from the viewpoint e to v, the normal cone is back-facing if:

$$\gamma < 90 \Rightarrow \cos\gamma > \cos(90 - \theta) \Rightarrow \cos\gamma > \sin\theta \qquad (4)$$

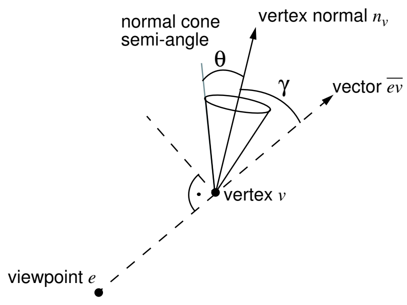


Fig. 25. Back-face simplification.

As mentioned at the beginning of this section, $\sin\theta$ is stored with each half-edge collapse instead of the actual semi-angle θ of the normal cone. The $\cos\gamma$ term can be computed by $n_v \cdot (v - e)/|v - e|$, the dot product of the normalized vertex normal n_v and the vector $\bar{ev}$ from e to v divided by the length of $\bar{ev}$. Note that $\bar{ev}$ has already been computed for the view-frustum simplification and can be reused for back-face culling.

Note that Equation 4 itself is not a conservative heuristic but the correct inequality for back-face testing of a normal cone. Furthermore, since our bounding normal cones are correctly computed for each half-edge collapse, this back-face simplification criterion is not an approximation as it is the case in [19]. The proposed heuristic is as good as the bounding normal cone hierarchy is: more exact for tight bounding cones, and conservative for overestimated normal cones. In any case, it is guaranteed that no vertex is removed that has a silhouette or front-facing surface normal.

C. Screen projection

In FastMesh we bound the projected area of triangles affected by a half-edge collapse using its bounding sphere with radius r, bounding normal cone with semi-angle θ, and the normalized vertex normal n_v. Let us call this set of triangles the domain t of t. For better understanding and simple graphical representation, Figure 26 shows the situation of projecting a back-facing surface area onto the view plane. At run-time, only front-facing regions have to be tested for screen projection but these are handled analogously after inverting the vertex normal n_v.

The visible area of t is bounded by the bounding sphere and thus approximated by πr^2. Furthermore, it is maximally visible from the viewpoint e if it is oriented perpendicularly to the projection direction $\bar{ev}$. Thus for a given surface normal n_v, the visible area of t is directly proportional to the cosine of the angle γ formed between $\bar{ev}$ and n_v (maximal if $\gamma = 0 \Rightarrow \cos\gamma = 1$, and minimal if $\gamma = 90 \Rightarrow \cos\gamma = 0$, see also Figure 26). However, due to the normal variation bounded by the bounding cone angle θ, we have to consider the area of t oriented at angles $\gamma \pm \theta$, with $\gamma - \theta$ being the worst case. Given that $\gamma \leq 90$ and $\gamma \geq \theta$ the maximal visible area of t is directly proportional to $\cos(\gamma - \theta)$. Screen projection is performed by perspective division, and therefore, the projected area on the view plane at distance d from the viewpoint can be bounded by:

$$\cos(\gamma - \theta) \cdot \frac{\pi r^2}{|v - e|^2} d^2 \qquad (5)$$

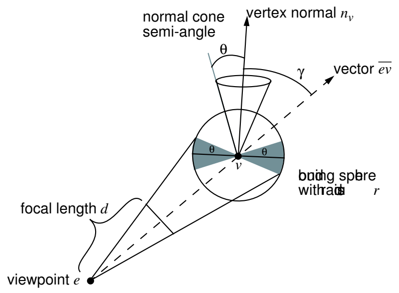


Fig. 26. Simplification based on screen projection.

The computation of the projected area according to Equation 5 would require an expensive calculation of the explicit value of γ, followed by another expensive cosine of $\gamma - \theta$. We avoid such costly trigonometric functions by the use of the trigonometric equality $\cos(\alpha) = \sin(90 - \alpha)$, and inequality $\sin(\alpha + \beta) \leq \sin\alpha + \sin\beta$, to get $\cos(\gamma - \theta) \leq \cos\gamma + \sin\theta$. Thus the projected area can be bounded by:

$$(\cos\gamma + \sin\theta) \cdot \frac{\pi r^2}{|v - e|^2} d^2 \qquad (6)$$

Therefore, a half-edge with bounding sphere radius r and normal cone with semi-angle θ can be collapsed if the projected area estimate of Equation 6 is smaller than the given threshold τ. For $\gamma \leq 90$ the cosine term $\cos\gamma$ can be reused from the back-face simplification test, otherwise if $\gamma > 90$ we can invert the vertex normal n_v and recalculate it by $\cos\gamma = -n_v \cdot (v - e)/|v - e|$. Note that also the length $|v - e|$ has already previously been computed and can be reused here as well.

For $\gamma \leq 90$ and $\gamma \geq \theta$ the worst case overestimation using $\cos\gamma + \sin\theta$ instead of $\cos(\gamma - \theta)$ would occur exactly when $\gamma = \theta$. However, we clamp $\cos\gamma + \sin\theta$ to be less or equal to 1.0 all the time because $\cos(\gamma - \theta) \leq 1.0$, and then the worst case factor occurs at $\gamma = 60$ and $\theta = 30$ with less than 16% overestimation, $\cos\gamma + \sin\theta < 1.16 \cdot \cos(\gamma - \theta)$. For any angles $\gamma < \theta$ we can set $\cos\gamma + \sin\theta$ strictly to 1.0.

D. Shading

We can measure the potential deviation in diffuse shading for a particular half-edge collapse operation by the variance in surface normals of the affected triangles. Therefore, we can bound the variation in shading by a function of the variance in normal directions, thus a function of the normal cone with semi-angle θ. In general the simplified surface will exhibit more shading artifacts for larger θ. Additionally the angle γ of the surface normal with respect to the projection direction may also be taken into account. For diffuse illumination the shading variation is proportional to θ, see also Figure 27.

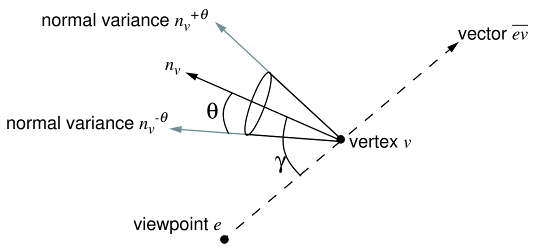

normal variance $n_v^{+\theta}$
vector $\overline{ev}$
n_v
θ
normal variance $n_v^{-\theta}$
vertex v
γ
viewpoint e

Fig. 27. Shading based simplification.

For a given angular normal deviation threshold φ – surface normal directions are allowed to change by up to φ degrees, all ecol operations with normal cone semi-angles $\theta < \varphi$ can be collapsed. Since we only maintain $\sin\theta$ for each half-edge collapse, we can precompute $\sin\varphi$ before dynamically adjusting the mesh, and test

$$\sin\theta < \sin\varphi \tag{7}$$

for each active node in the half-edge collapse hierarchy.

While this shading heuristic as presented above is not dependent on the actual viewpoint, it is a simplification criterion that affects the shading accuracy of the triangulated surface. The angular deviation tolerance $\theta < \varphi$ could easily be coupled with γ by $\theta < f(\varphi, \gamma)$ such that for smaller γ less angular deviation is tolerated than for larger γ. The result of this test, if it is not a function $f(\varphi, \gamma)$, could be computed on the entire hierarchy once for any given tolerance φ and stored as a boolean flag with each node. However, this flag would still be subject to testing at run-time since this visual simplification criterion is evaluated last among all visual error heuristics presented in this section. The hierarchy cannot be trimmed for this test in any way in a preprocess. Therefore, Equation 7 can be used at run-time without any drawback in performance or storage cost.

E. Silhouettes

A node t in the half-edge collapse hierarchy H is considered to contain the silhouette if its normal cone contains normals of mesh elements that are front-facing as well as some that are back-facing with respect to a particular viewpoint. This is the case as shown in Figure 28 if the normal cone contains the exact silhouette normal which is perpendicular to the vector $\bar{ev}$. Therefore, as illustrated in Figure 28 given the normal cone semi-angle θ, the angle γ between the vertex normal and the vector $\bar{ev}$ from the viewpoint e to v, a node is considered to be part of the silhouette if $|90 - \gamma| < \theta$. Thus a node cannot be simplified if the following inequality holds:

$$|\sin(90 - \gamma)| < sin\theta \Rightarrow (\cos\gamma)^2 < (\sin\theta)^2 \tag{8}$$

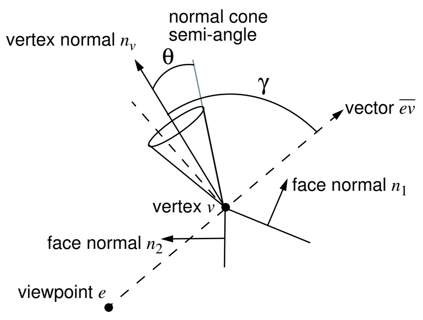


Fig. 28. Silhouette preservation.

The $\cos\gamma$ term is already computed for back-face simplification, thus can be reused, and $\sin\theta$ is stored with each half-edge collapse. The presented silhouette preservation test is only conservative in the sense that bounding normal cones conservatively represent the variation in normal directions, and it prevents removal of any vertex that is part of a silhouette edge on the highest mesh resolution.

Strict silhouette preservation can easily be relaxed by combining it with an edge-length tolerance or the screen projection heuristic described in the following section. Thus if a node is considered to be a silhouette according to the criteria mentioned above but if its edge length or screen projection is smaller than a given threshold it can still be allowed to be simplified.

Appendix II
Rendering Algorithms

```
void viewTestNode(bintree *node) {
   int merge;
   merge=viewTest(node); // perform view-dependent tests
   if (faces[node->edge/3].flag == COLLAPSED)
      if (!merge)
         split(node);
         if (node->l)
            viewTestNode(node->l);
         if (node->r)
            viewTestNode(node->r);
   else
      if (merge)
         collapse(node);
      else
         if (node->l)
            viewTestNode(node->l);
         if (node->r)
            viewTestNode(node->r);
}
```

Algorithm 1 Testing a node of the active nodes front.

```
int viewTest(bintree *node) {
   float $\overline{v}$[3], $\overline{ev}$[3], len, dot,
         cos$\gamma$, cos$\gamma^2$, sin$\theta^2$, factor, parea;

   // get vector $\overline{ev}$ from current viewpoint
   $\overline{ev}$ = $\overline{v}$ - $\overline{eye}$;      // v is start-vertex of node->edge
   len = sqrt($\overline{ev}$ · $\overline{ev}$);

   // get cosine of angle between $\overline{ev}$ and $\overline{dir}$
   dot = ($\overline{ev}$ · $\overline{dir}$);
   cos$\gamma$ = dot / len;

   // check if mesh element is outside of view frustum
   if (len > node->radius && cos$\gamma$ < cos$\omega$)
      if (dot < 0.0) // behind the viewpoint
         if (dot < -node->radius)
            return 1;
         if (len > node->radius/cos$\omega$)
            return 1;
      else if (cos$\gamma$ + node->radius/len < cos$\omega$)
         return 1;

   // get other variables
   cos$\gamma$ = ($\overline{n}_v$ · $\overline{ev}$) / len; // $\overline{n}_v$ is vertex normal
   cos$\gamma^2$ = cos$\gamma$ * cos$\gamma$;
   sin$\theta^2$ = node->sin$\theta$ * node->sin$\theta$;

   // back-face simplification
   if (node->sin$\theta$ < 1.0 && cos$\gamma$ > node->sin$\theta$)
      return 1;

   // do not simplify in silhouette areas
   if (cos$\gamma^2$ < sin$\theta^2$)
      return 0;

   // screen projection
   cos$\gamma$ = |cos$\gamma$|;
   if (cos$\gamma^2$ < 1.0 - sin$\theta^2$)
      factor = cos$\gamma$ + node->sin$\theta$;
      if (factor > 1.0)
         factor = 1.0;
   else
      factor = 1.0;
   parea = factor*$\pi$*node->radius$^2$*d$^2$/len$^2$;
   if (parea < $\tau$)
      return 1;

   // shading
   if (node->sin$\theta$ < sin$\varphi$)
      return 1;

   return 0;
}
```

Algorithm 2 Evaluating the view-dependent error heuristics.

```
void renderMesh() {
   int i;

   // render initial faces
   for (i = init; i < ne; i += 3)
      drawTriangle(hedges[i].vtx,
            hedges[i+1].vtx,
            hedges[i+2].vtx);

   // render detail faces, recursively render each root node
   for (i = 0; i < nr; i++)
      renderTree(merge[i]);
}

void renderTree(bintree *node) {
   int t, i;

   // stop traversal on leafs and collapsed nodes
   if (!node || collapsed(node))
      return;

   // render both faces corresponding to node
   t = tree - merge; // get index of node
   i = t * 6;
      drawTriangle(hedges[i].vtx,
            hedges[i+1].vtx,
            hedges[i+2].vtx);
   i = i + 3;
      drawTriangle(hedges[i].vtx,
            hedges[i+1].vtx,
            hedges[i+2].vtx);

   // recursively render chld nodes
   renderTree(lchild[t]);
   renderTree(rchild[t]);
}
```

Algorithm 3 Rendering of active faces.