In *Proceedings Pacific Graphics 2001*, pages 22–30. IEEE Computer Society Press, 2001.

FastMesh: Efficient View-dependent Meshing

Renato Pajarola[1]
Information & Computer Science
University of California Irvine
Irvine, CA 92697

Abstract

In this paper we present an optimized view-dependent meshing framework for adaptive and continuous level-of-detail (LOD) rendering in real-time. Multiresolution triangle mesh representations are an important tool for adapting triangle mesh complexity in real-time rendering environments. Ideally for interactive visualization, a triangle mesh is simplified to the maximal tolerated perceptual 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 an optimized computation of several view-dependent error metrics within that framework providing conservative error bounds. The presented approach called FastMesh, is highly efficient both in space and time cost, and it spends only a fraction of the time required for rendering to perform the error calculations and dynamic mesh updates.

CR Categories: I.3.3 [Computer Graphics]: Image Generation - Display algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - Surface and object representations.

Keywords: level-of-detail, multiresolution modeling, mesh simplification, interactive rendering

1. Introduction

Increasingly complex polygonal models exist today ranging from detailed CAD/CAM representations, high-resolution isosurfaces, extensive terrain surface models, large virtual environments, to complex digitized shapes (see also [LPC+00]). Such large models are hard to render at interactive frame rates due to the exceedingly large number of triangles. Level-of-detail (LOD) based visualization techniques [FS93] allow rendering the same object using triangle meshes of variable complexity. Thus the mesh complexity can dynamically be adjusted according to the object's relative position and importance in the rendered scene, and the complexity can also be adapted to guarantee stable interactive frame rates. Several mesh simplification and multiresolution triangulation methods [DP95, HG97, CMS98, LT99] have been developed to create different LODs, sequence of LOD-meshes with increasing complexity, and hierarchical triangulations for LOD based rendering.

Ideally for rendering, a triangle mesh is simplified to the maximal tolerated perceptual error that can be distinguished on screen. While finding this minimal 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 mesh simplification efficiently:

1. Parts of the surface outside of the visible area can be simplified to the maximum. 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 to the maximum. This reduces the amount of work for *back-face culling* in the rendering pipeline.
3. *Silhouettes* should be preserved due to their visual importance.
4. Surface regions with a very small projected area on screen may be represented with fewer triangles. Simplification can be performed until a user specified area *screen-projection* tolerance is reached.

Real-time rendering of a triangulated surface exhibits an extremely high frame-to-frame coherence of the rendered triangles. Thus 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 over time. Therefore, to take advantage of processing only the minimal number of mesh elements the simplification has to be updated incrementally from frame to frame. 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 4 listed above changes locally. In fact, only this incremental update makes simplification according to the properties 1 and 2 really useful.

While a lot of work has been done on mesh simplification only a few approaches have addressed the problem of view-dependent simplification for real-time rendering [XV96, Hop97, LE97], and performance optimization was not the main focus. In this paper we present a view-dependent meshing method called *FastMesh* with optimized data structures and algorithms for efficiently maintaining a dynamically simplified mesh using minimal amount of storage and data structures, and short preprocessing time. Furthermore, we present an optimized implementation with low CPU cost for the four view-dependent simplification criteria listed above. The two main contributions of this paper are:

- A hierarchical, half-edge data structure based multiresolution triangulation framework for view-dependent LOD rendering.
- Optimized computation and implementation of view-dependent error metrics for mesh simplification with conservative error bounds.

2. Related Work

2.1 Mesh simplification

Numerous methods for mesh simplification have been developed in the last decade, and a discussion is beyond the scope of this paper. For an overview on the various mesh simplification methods see [HG97], [CMS98], or [LT99]. Here we want to highlight the work of [Hop96] on progressive simplification of meshes, and [GH97] on simple geometric error measures. In [Hop96] a sequence of *n edge collapse* (ecol) operations is applied to simplify an arbitrary mesh M^n to a much 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 *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 Figure 1 that collapses a directed half-edge $\overrightarrow{v_1 v_2}$ to its endpoint v_2.

1. pajarola@acm.org, http://www.ics.uci.edu/~graphics/, P (949) 824 6357, F (949) 824 4056

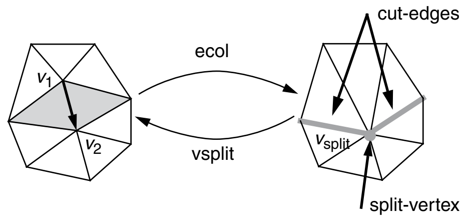

FIGURE 1. Half-edge collapse (ecol) and vertex split (vsplit) operations for triangle mesh simplification and refinement.

To create an initial set of ecol simplification operations we use the *quadric error metric* introduced in [GH97]. 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 computed by a matrix multiplication $(p^T \cdot v)^2 = v^T \cdot K_p \cdot v$ where the 4x4 quadric K_p is defined by the plane parameters as shown in [GH97]. Furthermore, the sum of squared distances to a set of planes P is then computed by $d(v, P)^2 = \sum_{p \in P} v^T \cdot K_p \cdot v = v^T \cdot \sum_{p \in P} K_p \cdot v$. Thus a set of planes can be represented by one quadric $Q_P = \sum_{p \in P} K_p$. In our implementation we additionally normalize the coefficients of Q_P by the number of planes $|P|$. Each vertex v is assigned a quadric Q^v that is initialized to the planes of the incident triangles. For a half-edge collapse $\overrightarrow{v_1 v_2}$ the quadrics of the collapsed vertices are added $Q = Q^{v_1} + Q^{v_2}$, and the approximation error introduced by that edge collapse is estimated by $v_2^T \cdot Q \cdot v_2$ (edge $\overrightarrow{v_1 v_2}$ is collapsed to v_2).

2.2 View-dependent meshes

Three methods on view-dependent meshing related to FastMesh have previously been proposed. We will briefly review and compare these approaches below.

In [XV96] a binary vertex hierarchy based on ecol and vsplit operations is constructed in a preprocess that allows interactive selective refinements at run-time. A very simple error metric using edge length is used to select the ecol operations at initialization. At run-time a front in the vertex hierarchy defines the currently visible mesh, and this front is interactively adjusted according to the current view point. The front is moved up and down in the hierarchy based on the screen projections of the edge lengths. In [XEV97] the approach is extended to consider surface normal information.

A similar approach to [XV96] is presented in [Hop97] where a sequence of ecol operations as proposed in [Hop96] is used to construct a binary vertex hierarchy. Furthermore, two image-space visibility attributes are implemented, and a screen-space geometric approximation error metric is proposed to modify the vertex front interactively.

In contrast to [XV96] our run-time simplification actually computes the four image-space error metrics outlined in the introduction in real-time using the same amount of storage – two scalar values per node. Furthermore, our half-edge collapse hierarchy imposes much fewer dependencies on the ecol and vsplit operations, and causes no storage cost thereof. Compared to [Hop97] our approach generates a more balanced half-edge collapse hierarchy due to selecting sets of independent ecol operations to create the hierarchy. Nevertheless, our method imposes even fewer restrictions for simplifying the mesh at run-time, thus provides a better view-dependent mesh adaptivity. We also introduce additional and more efficient image-space error metrics that require less storage and are faster to compute.

Our data structures do not require any extra storage for dependencies between vsplit and ecol operations, and only require two scalar values to be stored with each node in the half-edge collapse hierarchy to compute the view-dependent error metrics. Therefore, and due to the use of a half-edge data structure for dynamically maintaining the mesh connectivity, the data structures are extremely compact. The complete FastMesh representation uses significantly less memory (also on disk) than the previous approaches. Moreover, based on vertex position, normal, and only two scalar values per ecol operation we present a highly efficient implementation of all four image-space error metrics defined in the introduction.

A generalized framework on the basis of vertex hierarchies created from any kind of vertex contraction is presented in [LE97]. While this approach is very general it uses much more storage, and does not provide highly optimized calculations of view-dependent error metrics. While the method in [LE97] can be viewed as the least restricted generalization of view-dependent meshing, our approach provides best optimized performance with visibility culling integrated into the mesh simplification step.

3. Dynamic Meshing

In this section we describe the data structures and algorithms for dynamically maintaining a view-dependent mesh using a half-edge data structure [Wei85]. The use of a half-edge data structure makes the dynamical management of a constantly changing triangle mesh connectivity very efficient.

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 represented by an ordered set of three oriented half-edges. Every half-edge h stores information on its reverse *twin* half-edge (*h.r*), the next (*h.n*) and previous (*h.p*) half-edges in the triangle, and the starting vertex (*h.v*) of the half-edge as shown in Figure 2.

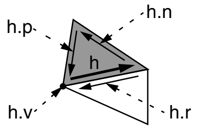

FIGURE 2. Half-edge data structure.

FastMesh stores a mesh of m triangles as an array of $3 \cdot m$ half-edges, and each group of three consecutive half-edges defines a triangle. In this representation the previous *h.p* and next *h.n* fields of a half-edge h with index i_h do not have to be actually stored, and can be computed efficiently by integer division (DIV) and modulo (MOD) operations, or by a simple conditional statement, but we will keep the notation *h.n* and *h.p* for simplicity.

The connectivity of the triangle mesh can efficiently be updated for a half-edge collapse, see also Figure 3. Collapsing the half-edge h and removing its incident triangles from the list of rendered triangles, requires the reverse information of the affected half-edges a, b, c and d to be updated such that the triangles A and B as well as C and D share an edge as shown in Figure 3 on the right. For triangles A and B, and given the half-edge h to be collapsed this can efficiently be done by:

$$h.p.r.r = h.n.r$$
$$h.n.r.r = h.p.r$$

Two similar assignments are required to setup the reverse information between triangles C and D. Note that the entries of all half-edges of the two triangles incident on h are not altered in any way, these triangles are just marked as not being used for rendering. Thus the inverse vsplit operation can reuse that information, and only the index i_h of the collapsed half-edge h has to be known for a vsplit operation.

In addition to updating the reverse information, all half-edges that have the same start-vertex *h.v* as the collapsed half-edge h need their start-vertex to be reassigned to the end-vertex of h which is *h.n.v*.

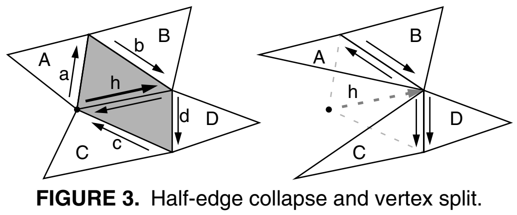

FIGURE 3. Half-edge collapse and vertex split.

Given a collapsed half-edge h by its index i_h, a vsplit operation must update the mesh connectivity to include the two triangles originally incident upon h. Note that the entries in the half-edge table for these two triangles have not been altered after collapsing h, and the incident faces A, B, C and D are valid faces in the current mesh as shown in Figure 3 on the right (see also preconditions on ecol and vsplit operations in Section 3.3). Therefore, for triangles A and B the connectivity can be restored by reverse-edge reassignments:

$$h.p.r.r = h.p$$
$$h.n.r.r = h.n$$

Similarly triangles C and D can efficiently be updated. Furthermore, all half-edges incident on the split-vertex, and between and including triangles A and C that currently used *h.n.v* as a start vertex, now have to be reassigned to *h.v*. Note that *h.v* is indeed the correct start vertex because nothing has been changed for h since its collapse.

3.2 Vertex hierarchy

Conceptually, a sequence of ecol operations computed during a preprocessing step defines a binary hierarchy as shown in [XV96] and [Hop97], and a view-dependent mesh is defined by a *front* through this hierarchy. However, the hierarchy in FastMesh differs significantly in implementation and semantics from these previous approaches.

To reduce storage cost, FastMesh defines the hierarchy H on half-edge collapses 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 the vertex based representations. Furthermore, this half-edge collapse hierarchy as shown in Figure 4 is implemented as a separate binary tree data structure, not merged with the vertex data, and only stores additional information per node that is required for the view-dependent error metrics. A node $t \in H$ consists of pointers to a parent node *t.p*, left *t.l* and right *t.r* child nodes, and an index $t.i_h$ of a collapsed edge h, plus two scalar values to compute the view-dependent error metrics (see also Section 4).

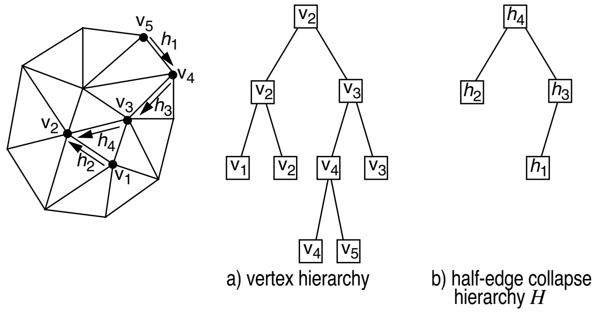

FIGURE 4. Binary half-edge collapse hierarchy H.

This definition of the binary half-edge collapse hierarchy H completely changes the semantics of the front $F \subseteq H$ through the hierarchy that defines a particular mesh. Although F defines a particular LOD mesh based on which edges are currently collapsed, it does not contain all visible mesh elements (triangles or vertices) of the current mesh.

Definition In FastMesh the front F consists of all *active* nodes in H, see also Figure 5. A node t is defined to be active if and only if one of the following two properties holds:

1. $t.i_h$ 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. $t.i_h$ 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, and is implemented as a doubly-linked linear list. In fact, F contains exactly all nodes for the current LOD that can potentially be collapsed (nodes of F_1), or that must be checked for mesh refinement (nodes of F_2, and all child nodes of F_1, $C_{F_1} = \{t | p \equiv t.p \wedge p \in F_1\}$).

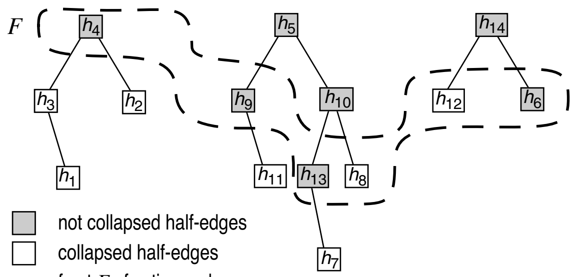

FIGURE 5. Front F of the current view-dependent mesh through the binary half-edge collapse hierarchy H.

At run-time, for every change in view parameters the front F is traversed and updated. First, nodes F_1 are tested to be collapsed. Second, all nodes F_2 and children C_{F1} of F_1 are tested to be split. All simplification and refinement operations have to be tested at run-time first to be *legal* as described in the following section, since they are performed out-of-order with respect to their global ordering at initialization.

3.3 Preconditions

A set of consecutive ecol operations can collapse multiple vertices to one vertex. Such a set of ecol operations forms a subtree in our half-edge collapse hierarchy, and must be performed bottom-up in correct partial order. An ecol operation is uniquely defined by an index i_h into the half-edge table, and any half-edge h that by collapsing may cause a topological singularity is not a legal half-edge collapse. A half-edge h is considered to be illegal if there exists a vertex V that is adjacent to both endpoints P and Q of h, and for which the three connected vertices P, Q, and V are not a triangle in the current mesh as shown in Figure 6.

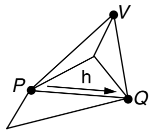

FIGURE 6. Topological ecol constraint. Half-edge h cannot be collapsed because of P and Q being connected to V.

Therefore, the half-edge h referenced from a node t is a legal candidate for an ecol operation only if:

1. no descendants of t have to be collapsed first,
2. and h is a topologically correct half-edge collapse.

Precondition 1 for ecol operations is satisfied by our definition of the front F of active nodes, and its subset F_1 that is tested for collapsing. Precondition 2 can be tested efficiently at run-time by examining the set of vertices V and edges e incident upon the endpoints of h. Condition 2 holds if:

$$\{V | \exists e: V = e.v \wedge e.n.v = h.v\} \cap$$
$$\{V | \exists e: V = e.v \wedge e.n.v = h.n.v\} \equiv \{h.p.v, h.r.p.v\}$$

Testing Precondition 2 involves visiting incident half-edges by rotation around both endpoints of h and testing for a non-empty intersection. This cost is small on average, but goes with $O(n^2)$ in the worst case.

A vertex split operation is also uniquely defined by an index i_h into the half-edge table, and vsplits must be performed partially ordered top-down in the hierarchy. The indexed collapsed half-edge h, and its two incident triangles contain all the required information to perform the vsplit operation as described in Section 3.1. However, the triangles A, B, C and D (referenced by *h.p.r*, *h.n.r*, *h.r.n.r* and *h.r.p.r*, see Figure 3) must currently be valid triangles in the half-edge data structure. Thus the half-edge h referenced by node t is a legal candidate for a vsplit operation only if:

1. all ancestors of t have been split,

2. and all four half-edges *h.p.r*, *h.n.r*, *h.r.n.r* and *h.r.p.r* pertain to valid faces in the current mesh.

Precondition 1 is met for all nodes F_2 and all children C_{F1} of nodes F_1 by definition of *F*. Precondition 2 is enforced by propagating vsplits to the nodes t_A, t_B, t_C and t_D, of *H* referenced by triangles *A*, *B*, *C* and *D*, before actually performing the vertex split of node *t*. However, note that this propagation can cause nodes to be split for which precondition 1 is not yet satisfied. Thus the recursive vsplit operations must first be propagated to the parent node *t.p* before recursively splitting nodes t_A, t_B, t_C and t_D.

To be able to propagate the vsplit operations as outlined above, each triangle face *A* must record the node t_A that causes the collapse of a half-edge *h* of triangle *A*, thus *h* is referenced by index $t_A.i_h$. Maintaining this information can be done dynamically at run-time whenever a half-edge *h* is actually collapsed.

4. View-dependent Error Metrics

In this section we describe the view-dependent error metrics outlined in Section 1. These error metrics have to be computed for all active nodes when the front *F* is traversed for updates as explained in Section 3.2. The nodes are collapsed or split based on: *view-frustum culling*, *back-face culling*, *silhouette preservation*, and *screen-projection tolerance*. Since the number of times these criteria must be computed per frame is in the order of the number of vertices in the current mesh, their computation must be extremely fast. FastMesh's view-dependent error metrics are designed to minimize computational costs, but nevertheless compute conservative error bounds on all four criteria.

As mentioned earlier, only two scalar values per node $t \in H$ in the half-edge collapse hierarchy are required to compute all four error metrics, and these two parameters are explained below.

Bounding sphere radius The bounding sphere centered at the endpoint of half-edge $t.i_h$ with radius *t.radius* encloses all triangles that are affected by the half-edge collapse *t* and all its descendants in *H* as shown in Figure 7. The bounding spheres can be computed bottom-up when building the hierarchy *H* at initialization.

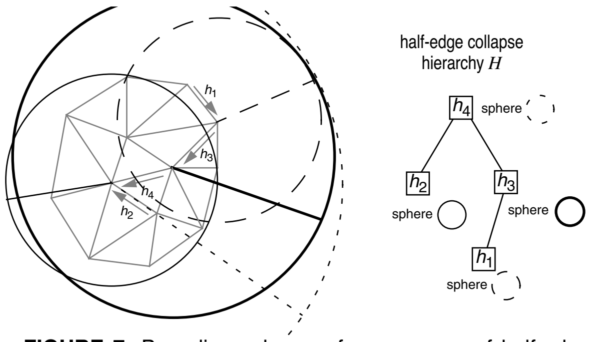

FIGURE 7. Bounding spheres of a sequence of half-edge collapse operations.

Normal cone angle The cone defined by the semi-angle θ (*t.theta*) about the vertex normal at the endpoint of half-edge $t.i_h$ bounds the cone of normals [SA93] of all triangles that are affected by the half-edge collapse *t* and all its descendants in *H* as shown in Figure 8. The bounding normal cones are also built bottom-up during the initialization process. Note that FastMesh actually only maintains the value of $\sin\theta$ instead of θ itself, since only $\sin\theta$ is needed to compute the view-dependent error metrics.

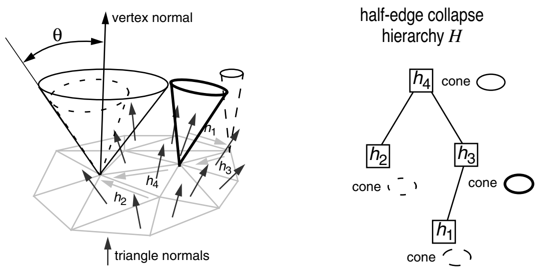

FIGURE 8. Bounding normal cone of a sequence of half-edge collapse operations.

4.1 View frustum

In order 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.

The distance to the view frustum can easily be computed given its four bounding planes as presented in [Hop97]. However, the 16 plane parameters must be calculated for every frame from the current view parameters since they are usually not given in a 3D graphics system. To avoid computation of the plane parameters 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, and are easily available in a 3D graphics system.

As shown in Figure 9, a half-edge collapse with endpoint *v* is outside the view frustum and can be performed if $d - a > b$. This can efficiently be computed if given the vectors *v*, *e*, *n* with $|n|=1$, the bounding sphere radius *r*, and the FOV semi-angle ω by

$$\begin{aligned} c &= n \cdot (v - e) \\ d &= |v - e - cn| \\ a &= c \tan\omega \\ b &= r / \cos\omega \end{aligned}$$

Note that the trigonometric functions $\tan\omega$ and $\cos\omega$ only have to be computed once at initialization when the aperture angle 2ω of the FOV is specified (or whenever the user changes the viewing aperture).

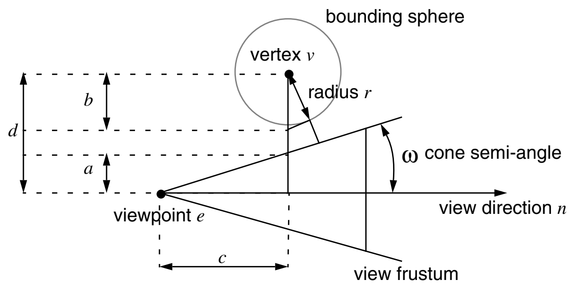

FIGURE 9. Outside view-frustum simplification.

An example view-frustum simplification is shown in Figure 14. Within the indicated view frustum, bounded by the yellow planes, the triangle mesh is rendered in full resolution, also the invisible parts, and the remaining parts of the mesh 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, or if rendered, are not visible to the viewer because obscured by other triangles. These unnecessary triangles can cost a significant amount of computation time, even if only processed and discarded using back-face culling. Thus back-facing regions of the mesh can be kept at the coarsest possible resolution.

Therefore, a half-edge with endpoint *v* can be collapsed if its associated normal cone is back-facing, that is if no normal within the cone can be front-facing. As can be seen from Figure 10, given the normal cone with semi-angle θ and the angle γ between the normalized vertex normal n_v and the vector $\overline{ev}$ from the viewpiont *e* to *v*, this is the case if:

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

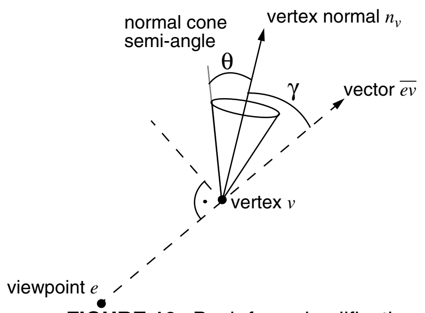

FIGURE 10. Back-face simplification.

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 $\overline{ev}$ from *e* to *v* divided by the length of $\overline{ev}$. Note that $\overline{ev}$ has already been computed for the view-frustum simplification and can be reused for back-face culling. Furthermore, since our bounding normal cones are correctly computed for each half-edge collapse, the back-face simplification criterion is not an approximation as it is the case in [Hop97].

Figure 15 shows an example for back-face simplification. Only the surface regions that are oriented towards the viewpoint are displayed in full resolution, back-facing areas are simplified as much as possible.

4.3 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.

An edge in a triangular mesh is defined to be a silhouette edge if one of the incident triangles is front-facing and the other is back-facing with respect to a particular viewpoint. In our multiresolution mesh we additionally have to consider the normal cone associated with a half-edge. Therefore, given the normal cone semi-angle θ, the angle γ between the vertex normal and the vector $\overline{ev}$ from the viewpiont *e* to *v*, and the face normals n_1 and n_2 of the incident triangles as shown in Figure 11, a half-edge is considered to be part of the silhouette, and thus cannot be collapsed, if one of the following inequalities holds:

$$|\sin(90 - \gamma)| < \sin\theta \Rightarrow (\cos\gamma)^2 < (\sin\theta)^2$$

$$(n_1 \cdot \overrightarrow{ev})(n_2 \cdot \overrightarrow{ev}) < 0 \text{ (with } \overrightarrow{ev} = v - e\text{)}$$

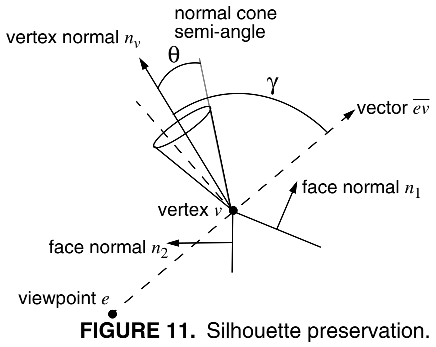

FIGURE 11. Silhouette preservation.

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 negative sign of the product of the dot-products of the triangle face normals n_1 and n_2 with vector $\overline{ev}$ determines if the normals have different orientations towards the viewpoint. Note that also $\overline{ev}$ has been computed previously and can be reused here without further computation.

The example in Figure 16 shows silhouette preservation nicely. While the triangle mesh has a high resolution along the silhouette area, other regions are simplified to the coarsest possible resolution.

4.4 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 geometry (transformation and lighting) limited and not pixel fill-rate limited, which is the case for most 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. Note that an ecol operation affects all triangles incident on the removed vertex. In FastMesh we can 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. For better understanding and simple graphical representation, Figure 12 shows the situation of projecting a back-facing surface area onto the view plane. The front-facing situation is handled analogously after inverting the vertex normal n_v.

With respect to a given viewpoint *e*, the visible area is maximal if $\gamma = 0° \Rightarrow \cos\gamma = 1$, and minimal if $\gamma = 90° \Rightarrow \cos\gamma = 0$, see also Figure 12. However, due to the normal variation bounded by θ, the maximal visible area for $\gamma < 90°$ can already occur when $\cos(\gamma - \theta)$ is 1. Thus for $\gamma \le 90° \Rightarrow \cos\gamma \ge 0$, $\gamma \ge \theta \Rightarrow \cos\gamma \le \cos\theta$, and given the area πr^2 of the intersection of the bounding sphere with a plane, the maximal visible area can be bounded by $\cos(\gamma - \theta) \cdot \pi r^2$. Therefore, the projected area on the view plane at distance *d* from the viewpoint can be bounded by:

$$\cos(\gamma - \theta) \cdot \frac{\pi r^2}{|v - e|^2} d^2 \qquad \textbf{(EQ 1)}$$

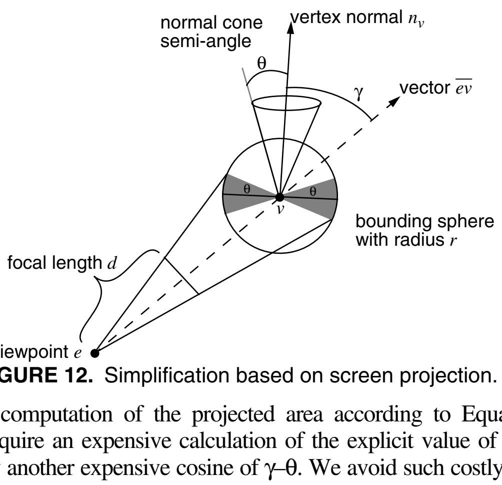

FIGURE 12. Simplification based on screen projection.

The computation of the projected area according to Equation 1 would require an expensive calculation of the explicit value of γ, followed by another expensive cosine of γ–θ. We avoid such costly trigonometric functions by the use of the trigonometric equality $\cos(\alpha) = \sin(90 - \alpha)$, and inequality $\sin(\alpha + \beta) \le \sin\alpha + \sin\beta$, to get $\cos(\gamma - \theta) \leq \cos\gamma + \sin\theta$. Thus for $\gamma \leq 90°$ and $\gamma \geq \theta$ the projected area can be bounded by

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

and a half-edge with bounding sphere radius *r* and normal cone with semi-angle θ can be collapsed if evaluation of Equation 2 is smaller than the given threshold τ.

For $\gamma < \theta$ we set $\cos\gamma + \sin\theta$ to 1. For $\gamma \leq 90°$ the cosine term can be reused from the back-face simplification, otherwise we can invert the vertex normal n_v and recalculate it from $\cos\gamma = -n_v \cdot (v - e)/|v - e|$. Note that also the length $|v - e|$ has already previously been computed and can be reused here.

Figure 17 shows an example with projected area error tolerance of $\tau = 1/2^{10} = 0.00098$, measured as a fraction (percentage) of the viewport size – red square of size τ on the imaginary unit viewing plane at the base of the transparently rendered view-frustum pyramid. The example shows nicely, but very subtle, the variance in simplification based on the distance from the viewpoint, and based on the angular orientation of the surface with respect to the view direction.

5. Initialization

The preprocessing, or initialization stage of the presented view-dependent mesh framework consists of the following steps:

1. Reading the triangle mesh input file, and creating the half-edge data structure.
2. Selecting a 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 and bounding normal cone semi-angle for each node $t \in H$.
4. Initialize the active front *F* to the coarsest mesh M^0.

For Step 2 above, a simplification process based on an object-space geometric error metric is appropriate since less important features in object-space will also likely to be of less visual importance in image-space. Also, the view-dependent simplification criteria cannot be used for the view-independent preprocess. We use the quadric error metric described in Section 2.1 to guide the preprocess simplification. Furthermore, to balance the hierarchy *H*, and to reduce dependencies of the partial ordering of nodes $t \in H$, the selection of simplification operations is performed in batches similar to [PR00].

Initializing the binary half-edge collapse hierarchy *H* in Step 3 involves setting the correct parent-child relations between the selected half-edge collapse operations. Additionally, the view-dependent error metric coefficients, 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*.θ 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,* respectively *t.r*.θ and *t.l*.θ of *t*'s children. Finally, all entries *t*.θ are converted to $\sin(t.\theta)$.

Note that Steps 1 to 3 can actually be performed once per object or triangulated surface, and the half-edge collapse hierarchy can be stored with the vertex and triangle mesh data as outlined in Section 7.1.

6. Implementation Details

6.1 Data structures and algorithms

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. A view-dependent multiresolution mesh with *n* vertices and *m* triangles consists of several arrays: vertex coordinates (float vertices[n][3]), vertex normals (float normals[n][3]), half-edges (halfedge hedges[3m]), and triangle faces (tface faces[m]). Additionally, it also includes the nodes of the binary half-edge collapse hierarchy, and at run-time the doubly linked list of active nodes. The main data structures are given in Figure 13 below.

```
struct halfedge {       // half-edge data structure
   int rev;
   int vertex;
};
struct tface {          // auxiliary information per face
   char flag;           // 0 if not currently used in the mesh
   bintree *split;      // node that deletes (inserts) this triangle
};
struct bintree {        // binary half-edge collapse hierarchy
   bintree *l,*r,*p;    // left, right, and parent links
   int edge;            // collapsed half-edge index
   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
};
```

FIGURE 13. FastMesh main data structures.

For each face we maintain a flag that specifies if that face is currently rendered or not, and a pointer to the corresponding node in the half-edge collapse hierarchy *H*. This is necessary to propagate forced splits as described in Section 3.3. A node of *H* consists of the parent-child links, and the collapsed half-edge index as explained in Section 3.2, and of the error metric coefficients introduced in Section 4. Given a half-edge with index *i*, its corresponding face is indexed by *i*/3 if both half-edge and face arrays are ordered consistently.

In the main rendering loop, the active nodes *F* have 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*. In Algorithm 1 below, the necessary steps for testing an active node $\in F$ are given. For each node, the tests described in Section 4 are evaluated, and if the node represents a collapsed half-edge it is refined if required by the error metric. Otherwise, the half-edge is collapsed if allowed by the error metric, or its children are tested to be split.

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

The procedure to evaluate the view-dependent error metrics is given below in Algorithm 2. The code has been modified from standard C++ to be more concise, it uses vector variables (i.e. $\overline{v}$) and vector operations (+, -, and dot product ·), and complex variable names such as cosγ or sinθ^2. 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 tanω and cosω), and focal length d.

```
int viewTest(bintree *node) {
   float $\overline{v}$[3], a, $\overline{q}$[3], qlen, $\overline{ev}$[3], len, dot,
      cosγ, cosγ$^2$, sinθ$^2$, factor, parea;

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

   // check if mesh element is in front of viewpoint
   if ((dot = $\overline{ev}$ · $\overline{dir}$) <= 0.0) return 1;

   // check if mesh element is in view frustum
   $\overline{q}$ = $\overline{ev}$ - dot * $\overline{dir}$;
   qlen = sqrt($\overline{q}$[0]$^2$ + $\overline{q}$[1]$^2$ + $\overline{q}$[2]$^2$);
   a = dot * tanω;
   if (qlen - a > node->radius * cosω) return 1;

   // get squared length of ev and other variables
   len = sqrt($\overline{ev}$ · $\overline{ev}$);
   cosγ = ($\overline{n}_v$ · $\overline{ev}$) / len;  // $\overline{n}_v$ is vertex normal
   cosγ$^2$ = cosγ * cosγ;
   sinθ$^2$ = node->sinθ * node->sinθ;

   // check for back-face simplification
   if (cosγ > node->sinθ) return 1;

   // preserve silhouette edges, $\overline{n}_1$ and $\overline{n}_2$ are adjacent face normals
   if (cosγ$^2$ < sinθ$^2$ || ($\overline{n}_1$·$\overline{ev}$)*($\overline{n}_2$·$\overline{ev}$) < 0.0)
      return 0;

   // check screen projection tolerance
   cosγ = |cosγ|;
   if (cosγ$^2$ < 1.0 - sinθ$^2$)
      factor = cosγ + node->sinθ;
      if (factor > 1.0) factor = 1.0;
   else
      factor = 1.0;
   parea = factor * π * node->radius$^2$ * d / len;
   if (parea < τ) return 1;

   return 0;
}
```

ALGORITHM 2. Evaluating view-dependent error metric.

6.2 Limitations

As presented in this paper, this view-dependent simplification method using a half-edge triangle mesh data structure, and a half-edge collapse hierarchy works well on manifold triangular meshes of arbitrary topology (genus). Altough the special cases are not discussed here, simplification near and on boundary vertices and edges of the mesh is possible with minor modifications.

Moreover, our method naturally handles meshes which have a manifold mesh connectivity but that may have non-manifold vertices only. Handling of non-manifold edges would require significant changes to the half-edge data structure, half-edge collapse operation, and its inverse vertex split operation.

7. Experimental Results

We tested our view-dependent meshing approach on various models of different sizes and varying shapes. The experiments include measurements of the compactness of the FastMesh data structures, as well as run-time performance of the view-dependent error calculations, and timing of the dynamic mesh updates using the half-edge collapse hierarchy. Graphical examples are also given in Figure 18 for a triangulated terrain surface.

7.1 Space cost

The main memory usage of FastMesh is determined by the number of mesh elements, and the size of the data structures of Figure 13. Note that the number of nodes in the half-edge collapse hierarchy is smaller than the number of vertices. For a mesh with *n* vertices, and thus about 2*n* triangles and 6*n* half edges, the run-time space cost consists of: 24*n* bytes for vertex coordinates and normals, 5·2*n* bytes for the faces, 8·6*n* bytes for the half edges, and 24*n* bytes for the half-edge hierarchy. Thus the overall main memory size is only 106*n* bytes, see also Table 1. The main space cost advantage is due to the fact that the half-edge collapse hierarchy only requires half the number of nodes compared to a binary vertex hierarchy, and due to the compact vertex, face, and edges data structures.

Storage of a FastMesh on disk requires even less space. Note that the faces data as well as the reverse fields of half-edges can be recovered at initialization, and need not to be stored. Thus since the binary hierarchy can also be recovered by the collapsed half-edge index of each node, the disk space reduces to 24*n* (coordinates and normals) + 4·6*n* (half edge start vertices) + 12*n* (node half-edges indices, and error metric coefficients) = 60*n* bytes.

Model	Full resolution		FastMesh		
	vertices	faces	memory	disk	gzipped
bunny	35947	69451	3.5	2.0	1.5
fandisk	6475	12946	0.7	0.4	0.2
happy	49794	100000	5.0	2.9	2.0
horse	48485	96966	4.9	2.8	1.9
phone	83044	165963	8.4	4.8	3.4
terrain	90000	178802	9.0	5.1	2.5

TABLE 1. Space cost of the FastMesh data structure in main memory, on disk, and gzipped on disk given in MBytes.

7.2 Time cost

The run-time performance tests were performed on a Sun Ultra60 workstation, equipped with a 450MHz UltraSPARC-II CPU, an Expert3D PCI-bus graphics card, and running SunOS 5.7. The CPU time usage was measured with the high-resolution timing function gethrvtime() in conjunction with the ptime command for microstate CPU accounting available on Sun/Solaris machines.

Table 2 in the left-hand columns reports construction time of a FastMesh data structure when initialized from a plain triangle mesh (columns *ecol selection* and *hierarchy*) as well as initialization from a FastMesh representation stored on disk (column *from disk*). Initialization from a plain triangle mesh 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 error metric coefficients as described in the introduction of Section 4. Initialization from disk consists of reading and reconstructing the half-edge data structure, as well as reading the error metric coefficients and reconstructing the binary half-edge hierarchy from the sequence of half-edge collapses.

Model	Initialization			Run-time (averaged per frame)								
	ecol selection	hierarchy	from disk	rendering			error metric			updating mesh		
				$\lvert\Delta\rvert$	time	% of frame	|tests|	time	% of frame	|updates|	time	% of frame
bunny	10s	0.4s	1.4s	15402	23.5 ms	44%	5423	8.9 ms	17%	1348	4.1 ms	8%
fandisk	1.7s	0.06s	0.3s	3790	3.9 ms	35%	1242	1.8 ms	16%	294	0.7 ms	6%
happy	20s	0.6s	7.4s	22487	31.8 ms	33%	7584	14.4 ms	15%	2101	7.0 ms	7%
horse	15s	0.6s	2.2s	18889	25.7 ms	34%	6708	12.1 ms	16%	2719	8.8 ms	12%
phone	26s	1.1s	3.4s	31175	60.7 ms	38%	10735	20.9 ms	13%	4522	16.6 ms	10%
terrain	40s	1.1s	17s	25185	89.1 ms	46%	9314	16.7 ms	9 %	2771	9.2 ms	5%

TABLE 2. Initialization performance given in seconds to preprocess a given triangle mesh and constructing the half-edge hierarchy, or read a FastMesh data structure from disk. Run-time performance for rendering, computing error measures, and updating the triangle mesh data structure is given by the average number of elements processed, the time in milliseconds to perform each task, and its percentage of overall CPU cost per frame.

Run-time performance was also measured in CPU time usage using gethrvtime() and ptime. The three tasks that were timed are:

1. Rendering, which includes setting up the graphics context and calls to the OpenGL GLvertex() and GL normal() functions.
2. Calculating and testing the view-dependent error metric for active nodes.
3. Updating the mesh using vertex split and half-edge collapse operations according to the results of the view-dependent tests on active nodes.

For each task we counted the number of elements that were processed, and measured the CPU time that was used. The test run consisted of moving the view-frustum continuously around the object, and constantly varying the threshold τ.

Table 2 presents the achieved results averaged per frame, and shows the run-time cost of each individual task in relation to the overall CPU cost. As can be seen, by far most of the CPU time is spent on the rendering task. Even though a significant number of view-tests has to be performed each frame, our approach is very efficient and only consumes a small fraction of the overall time to perform these tests. Furthermore, also the dynamically changing half-edge based triangle mesh data structure is extremely efficient, it mostly consumes less than 10% of the per frame CPU cost. The computation of the view-dependent error metrics for a single half-edge collapse only requires less than 2μs, and an individual ecol or vsplit mesh update operation needs about 2.4 to 3.7μs on average.

8. Conclusion

In this paper we have presented an efficient view-dependent meshing framework 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 using this half-edge hierarchy, and devised a set of effective view-dependent error metrics and a highly efficient implementation to guide mesh simplification. Experiments on a variety of meshes have shown the efficiency of the presented view-dependent error metric calculations, as well as the compactness of the required data structures. Compared to previous methods, our approach exhibits much fewer dependencies between refinement and simplification operations, uses simpler and more compact data structures, and integrates highly optimized calculations of several view-dependent error metrics.

Future work related to this area includes view-dependent simplification of non-manifold triangle meshes and triangle soups, dynamic triangle strip generation, and out-of-core view-dependent triangulation.

References

[CMS98] P. Cignoni, C. Montani and R. Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22(1):37–54, 1998.

[DP95] Leila De Floriani and Enrico Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Transactions on Graphics*, 14(4):363–411, 1995.

[FS93] Thomas Funkhouser and Carlo Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings SIGGRAPH 93*, pages 247–254. ACM SIGGRAPH, 1993.

[GH97] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings SIGGRAPH 97*, pages 209–216. ACM SIGGRAPH, 1997.

[HG97] Paul S. Heckbert, and Michael Garland. Survey of polygonal surface simplification algorithms. SIGGRAPH 97 Course Notes 25, 1997.

[Hop96] Hugues Hoppe. Progressive meshes. In *Proceedings SIGGRAPH 96*, pages 99–108. ACM SIGGRAPH, 1996.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. In *Proceedings SIGGRAPH 97*, pages 189–198. ACM SIGGRAPH, 1997.

[LPC$^+$00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, Duane Fulk. The digital Michelangelo project: 3D scanning of large satues. In *Proceedings SIGGRAPH 2000*, pages 131–144. ACM SIGGRAPH, 2000.

[LT99] Peter Lindstrom and Greg Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, April-June, 1999.

[LE97] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings SIGGRAPH 97*, pages 199–208. ACM SIGGRAPH, 1997.

[PR00] Renato Pajarola and Jarek Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, January-March 2000.

[SA93] Leon A. Shirman and Salim S. Abi-Ezzi. The cone of normals technique for fast processing of curved patches. In *Proceedings EUROGRAPHICS 93*, pages C261–C272, 1993. also in Computer Graphics Forum 12(3).

[Wei85] Kevin Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1): pages 21-40, January 1985.

[XV96] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In Proceedings Visualization 96, pages 327–334. IEEE, Computer Society Press, Los Alamitos, California, 1996.

[XEV97] Julie C. Xia, Jihad El-Sana and Amitabh Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, April-June 1997.

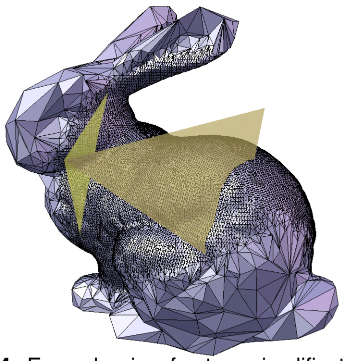

FIGURE 14. Example view-frustum simplification with two sides of a simulated view frustum shown as transparent yellow planes. Rendered 62%, or 43463 out of 69451 faces.

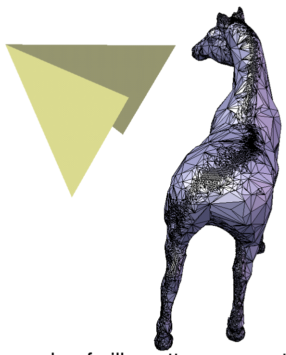

FIGURE 16. Example of silhouette preservation. Rendered 19%, or 19154 out of 96966 faces.

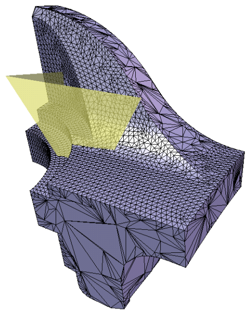

FIGURE 15. Example back-face simplification for the displayed view frustum. Rendered 51%, or 6684 out of 12946 faces.

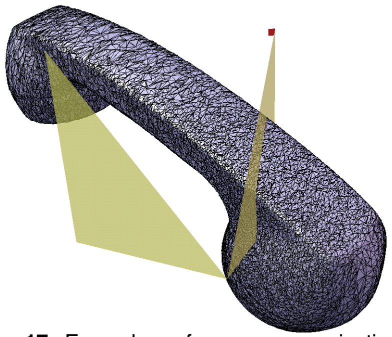

FIGURE 17. Example of screen projection based simplification with projection tolerance $\tau = 1/2^{10}$. Rendered 19%, or 32755 out of 165963 faces.

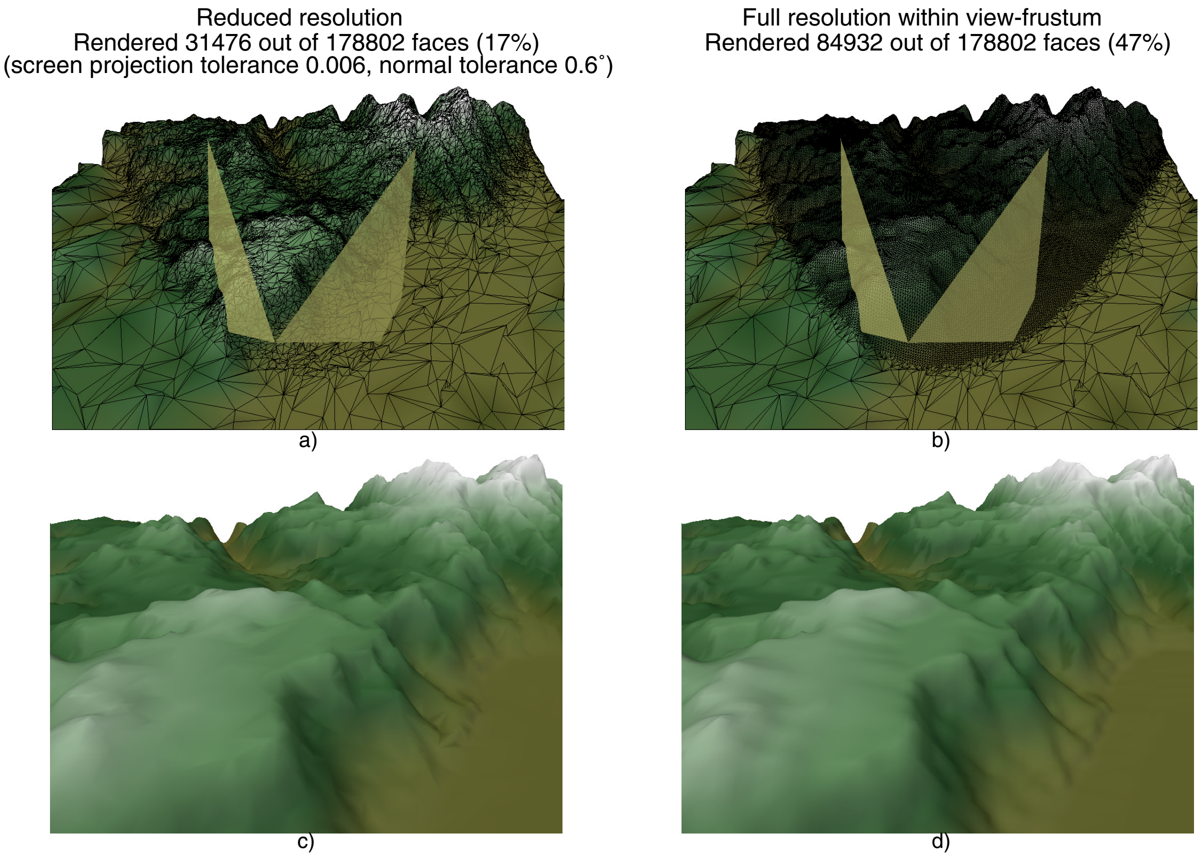
Reduced resolution
Rendered 31476 out of 178802 faces (17%)
(screen projection tolerance 0.006, normal tolerance 0.6˚)
Full resolution within view-frustum
Rendered 84932 out of 178802 faces (47%)
a)
b)
c)
d)

FIGURE 18. Images a) and b) show the view-frustum and mesh simplification from a bird's eye view, and images 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 show 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 squared length of the silhouette edge as projected on the view-plane and thresholding it with 0.01τ.