View-Dependent Refinement of Progressive Meshes

Hugues Hoppe
Microsoft Research

ABSTRACT

Level-of-detail (LOD) representations are an important tool for real-time rendering of complex geometric environments. The previously introduced *progressive mesh* representation defines for an arbitrary triangle mesh a sequence of approximating meshes optimized for view-independent LOD. In this paper, we introduce a framework for selectively refining an arbitrary progressive mesh according to changing view parameters. We define efficient refinement criteria based on the view frustum, surface orientation, and screen-space geometric error, and develop a real-time algorithm for incrementally refining and coarsening the mesh according to these criteria. The algorithm exploits view coherence, supports frame rate regulation, and is found to require less than 15% of total frame time on a graphics workstation. Moreover, for continuous motions this work can be amortized over consecutive frames. In addition, smooth visual transitions (geomorphs) can be constructed between any two selectively refined meshes.

A number of previous schemes create view-dependent LOD meshes for height fields (e.g. terrains) and parametric surfaces (e.g. NURBS). Our framework also performs well for these special cases. Notably, the absence of a rigid subdivision structure allows more accurate approximations than with existing schemes. We include results for these cases as well as for general meshes.

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

Additional Keywords: mesh simplification, level-of-detail, multiresolution representations, dynamic tessellation, shape interpolation.

1 INTRODUCTION

Rendering complex geometric models at interactive rates is a challenging problem in computer graphics. While rendering performance is continually improving, significant gains are obtained by adapting the complexity of a model to its contribution to the rendered image. The ideal solution would be to efficiently determine the coarsest model that satisfies some perceptual image qualities. One common heuristic technique is to author several versions of a model at various *levels of detail* (LOD); a detailed triangle mesh is used when the object is close to the viewer, and coarser approximations are substituted as the object recedes [4, 8]. Such LOD meshes can be computed automatically using mesh simplification techniques (e.g. [5, 10, 19, 21]). The recently introduced *progressive mesh* (PM) representation [10] captures a continuous sequence of meshes optimized for view-independent LOD control, and allows fast traversal of the sequence at runtime.

Sets or sequences of view-independent LOD meshes are appropriate for many applications, but difficulties arise when rendering large-scale models, such as environments, that may surround the viewer:

- Many faces of the model may lie outside the view frustum and thus do not contribute to the image (Figure 12a). While these faces are typically culled early in the rendering pipeline, this processing incurs a cost.
- Similarly, it is often unnecessary to render faces oriented away from the viewer, and such faces are usually culled using a "back-facing" test, but again at a cost.
- Within the view frustum, some regions of the model may lie much closer to the viewer than others. View-independent LOD meshes fail to provide the appropriate level of detail over the entire model (e.g. as does the mesh in Figure 12b).

Some of these problems can be addressed by representing a graphics scene as a hierarchy of meshes. Parts of the scene outside the view frustum can then be removed efficiently using hierarchical culling, and LOD can be adjusted independently for each mesh in the hierarchy [4, 8]. However, establishing such hierarchies on continuous surfaces is a challenging problem. For instance, if a terrain mesh (Figure 11d) is partitioned into blocks, and these blocks are rendered at different levels of detail, one has to address the problem of cracks between the blocks [14]. In addition, the block boundaries are unlikely to correspond to natural features in the surface, resulting in suboptimal approximations. Similar problems also arise in the adaptive tessellation of smooth parametric surfaces [1, 13, 18].

Specialized schemes have been presented to adaptively refine meshes for the cases of height fields and parametric surfaces, as summarized in Section 2.1. In this paper, we offer a general runtime LOD framework for selectively refining arbitrary meshes according to changing view parameters. A similar approach was developed independently by Xia and Varshney [24]; their scheme is summarized and compared in Section 2.3.

The principal contributions of this paper are:

- It presents a framework for real-time selective refinement of arbitrary progressive meshes (Section 3).
- It defines fast view-dependent refinement criteria involving the view frustum, surface orientation, and screen-space projected error (Section 4).
- It presents an efficient algorithm for incrementally adapting the mesh refinement based on these criteria (Section 5). The algorithm exploits view coherence, supports frame rate regulation, and may be amortized over consecutive frames. To reduce popping, geomorphs can be constructed between any two selectively refined meshes.

Email: hhoppe@microsoft.com
Web: http://research.microsoft.com/∼hoppe/

- It shows that triangle strips can be generated for efficient rendering even though the mesh connectivity is irregular and dynamic (Section 6).
- Finally, it demonstrates the framework's effectiveness on the important special cases of height fields and tessellated parametric surfaces, as well as on general meshes (Section 8).

Notation We denote a triangle mesh M as a tuple (V, F), where V is a set of vertices v_j with positions $\mathbf{v}_j \in \mathbf{R}^3$, and F is a set of ordered vertex triples $\{v_j, v_k, v_l\}$ specifying vertices of triangle faces in counter-clockwise order. The *neighborhood* of a vertex v, denoted N_v, refers to the set of faces adjacent to v.

2 RELATED WORK

2.1 View-dependent LOD for domains in $\mathbf{R}^2$

Previous view-dependent refinement methods for domains in $\mathbf{R}^2$ fall into two categories: height fields and parametric surfaces.

Although there exist numerous methods for simplifying height fields, only a subset support efficient view-dependent LOD. These are based on hierarchical representations such as grid quadtrees [14, 23], quaternary triangular subdivisions [15], and more general triangulation hierarchies [3, 6, 20]. (The subdivision approach of [15] generalizes to 2-dimensional domains of arbitrary topological type.) Because quadtrees and quaternary subdivisions are based on a regular subdivision structure, the view-dependent meshes created by these schemes have constrained connectivities, and therefore require more polygons for a given accuracy than so-called *triangulated irregular networks* (TIN's). It was previously thought that dynamically adapting a TIN at interactive rates would be prohibitively expensive [14]. In this paper we demonstrate real-time modification of highly adaptable TIN's. Moreover, our framework extends to arbitrary meshes.

View-dependent tessellation of parametric surfaces such as NURBS requires fairly involved algorithms to deal with parameter step sizes, trimming curves, and stitching of adjacent patches [1, 13, 18]. Most real-time schemes sample a regular grid in the parametric domain of each patch to exploit fast forward differencing and to simplify the patch stitching process. Our framework allows real-time adaptive tessellations that adapt to surface curvature and view parameters.

2.2 Review of progressive meshes

In the PM representation [10], an arbitrary mesh $\hat{M}$ is simplified through a sequence of n *edge collapse* transformations (*ecol* in Figure 1) to yield a much simpler base mesh M^0 (see Figure 11):

$$(\hat{M}=M^n) \xrightarrow{ecol_{n-1}} \ldots \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0 .$$

Because each *ecol* has an inverse, called a *vertex split* transformation, the process can be reversed:

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \ldots \xrightarrow{vsplit_{n-1}} (M^n=\hat{M}) .$$

The tuple $(M^0, \{vsplit_0, \ldots, vsplit_{n-1}\})$ forms a PM representation of $\hat{M}$. Each vertex split, parametrized as $vsplit(v_s, v_l, v_r, v_t, f_l, f_r)$, modifies the mesh by introducing one new vertex v_t and two new faces $f_l = \{v_s, v_t, v_l\}$ and $f_r = \{v_s, v_r, v_t\}$ as shown in Figure 1. The resulting sequence of meshes $M^0, \ldots, M^n = \hat{M}$ is effective for view-independent LOD control (Figure 11). In addition, smooth visual transitions (*geomorphs*) can be constructed between any two meshes in this sequence.

To create view-dependent approximations, our earlier work [10] describes a scheme for selectively refining the mesh based on a user-specified query function $\mathsf{qrefine}(v_s)$. The basic idea is to traverse the

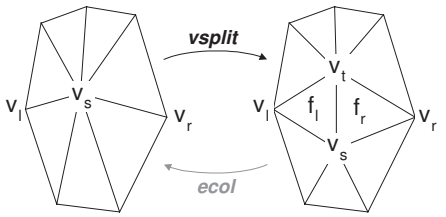


Figure 1: Original definitions of the refinement (*vsplit*) and coarsening (*ecol*) transformations.

$vsplit_i$ records in order, but to only perform $vsplit_i(v_{s_i}, v_{l_i}, v_{r_i}, \ldots)$ if

(1) $vsplit_i$ is a *legal* transformation, that is, if the vertices $\{v_{s_i}, v_{l_i}, v_{r_i}\}$ satisfy some conditions in the mesh refined so far, and

(2) $\mathsf{qrefine}(v_{s_i})$ evaluates to *true*.

The scheme is demonstrated with a view-dependent qrefine function whose criteria include the view frustum, proximity to silhouettes, and screen-projected face areas.

However, some major issues are left unaddressed. The qrefine function is not designed for real-time performance, and fails to measure screen-space geometric error. More importantly, no facility is provided for efficiently adapting the selectively refined mesh as the view parameters change.

2.3 Vertex hierarchies

Xia and Varshney [24] use *ecol*/*vsplit* transformations to create a simplification hierarchy that allows real-time selective refinement. Their approach is to precompute for a given mesh $\hat{M}$ a *merge tree* bottom-up as follows. First, all vertices $\hat{V}$ are entered as leaves at level 0 of the tree. Then, for each level $l \geq 0$, a set of *ecol* transformations is selected to merge pairs of vertices, and the resulting proper subset of vertices is promoted to level $l+1$. The *ecol* transformations in each level are chosen based on edge lengths, but with the constraint that their neighborhoods do not overlap. The topmost level of the tree (or more precisely, forest) corresponds to the vertices of a coarse mesh M^0. (In some respects, this structure is similar to the subdivision hierarchy of [11].)

At runtime, selective refinement is achieved by moving a vertex front up and down through the hierarchy. For consistency of the refinement, an *ecol* or *vsplit* transformation at level l is only permitted if its neighborhood in the selectively refined mesh is identical to that in the precomputed mesh at level l; these additional dependencies are stored in the merge tree. As a consequence, the representation shares characteristics of quadtree-type hierarchies, in that only gradual change is permitted from regions of high refinement to regions of low refinement [24].

Whereas Xia and Varshney construct the hierarchy based on edge lengths and constrain the hierarchy to a set of levels with non-overlapping transformations, our approach is to let the hierarchy be formed by an unconstrained, geometrically optimized sequence of *vsplit* transformations (from an arbitrary PM), and to introduce as few dependencies as possible between these transformations, in order to minimize the complexity of approximating meshes.

Several types of view-dependent criteria are outlined in [24], including local illumination and screen-space projected edge length. In this paper we detail three view-dependent criteria. One of these measures screen-space surface approximation error, and therefore yields mesh refinement that naturally adapts to both surface curvature and viewing direction.

Another related scheme is that of Luebke [16], which constructs a vertex hierarchy using a clustering octree, and locally adapts the complexity of the scene by selectively coalescing the cluster nodes.

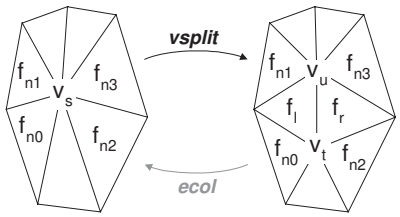


Figure 2: New definitions of *vsplit* and *ecol*.

3 SELECTIVE REFINEMENT FRAMEWORK

In this section, we show that a real-time selective refinement framework can be built upon an arbitrary PM.

Let a selectively refined mesh M^S be defined as the mesh obtained by applying to the base mesh M^0 a subsequence $S \subseteq \{0, \ldots, n-1\}$ of the PM *vsplit* sequence. As noted in Section 2.2, an arbitrary subsequence S may not correspond to a well-defined mesh, since a *vsplit* transformation is *legal* only if the current mesh satisfies some preconditions. These preconditions are analogous to the vertex or face dependencies found in most hierarchical representations [6, 14, 24]. Several definitions of *vsplit* legality have been presented (two in [10] and one in [24]); ours is yet another, which we will introduce shortly. Let $\mathcal{M}$ be the set of all meshes M^S produced from M^0 by a subsequence S of legal *vsplit* transformations.

To support incremental refinement, it is necessary to consider not just *vsplit*'s, but also *ecol*'s, and to perform these transformations in an order possibly different from that in the PM sequence. A major concern is that a selectively refined mesh should be unique, regardless of the sequence of (legal) transformations that leads to it, and in particular, it should still be a mesh in $\mathcal{M}$.

We first sought to extend the selective refinement scheme of [10] with a set of legality preconditions for *ecol* transformations, but were unable to form a consistent framework without overly restricting it. Instead, we began anew with modified definitions of *vsplit* and *ecol*, and found a set of legality preconditions sufficient for consistency, yet flexible enough to permit highly adaptable refinement. The remainder of this section presents these new definitions and preconditions.

New transformation definitions The new definitions of *vsplit* and *ecol* are illustrated in Figure 2. Note that their effects on the mesh are still the same; they are simply parametrized differently. The transformation $vsplit(v_s, v_t, v_u, f_l, f_r, f_{n0}, f_{n1}, f_{n2}, f_{n3})$, replaces the *parent* vertex v_s by two *children* v_t and v_u. Two new faces f_l and f_r are created between the two pairs of neighboring faces (f_{n0}, f_{n1}) and (f_{n2}, f_{n3}) adjacent to v_s. The edge collapse transformation $ecol(v_s, v_t, v_u, \ldots)$ has the same parameters as *vsplit* and performs the inverse operation. To support meshes with boundaries, face neighbors $f_{n0}, f_{n1}, f_{n2}, f_{n3}$ may have a special *nil* value, and vertex splits with $f_{n2} = f_{n3} = nil$ create only the single face f_l.

Let $\mathcal{V}$ denote the set of vertices in all meshes of the PM sequence. Note that $|\mathcal{V}|$ is approximately twice the number $|\hat{V}|$ of original vertices because of the vertex renaming in each *vsplit*. In contrast, the faces of a selectively refined mesh M^S are always a subset of the original faces $\hat{F}$. We number the vertices and faces in the order that they are created, so that $vsplit_i$ introduces the vertices $t_i = |V^0| + 2i + 1$ and $u_i = |V^0| + 2i + 2$. We say that a vertex or face is *active* if it exists in the selectively refined mesh M^S.

Vertex hierarchy As in [24], the parent-child relation on the vertices establishes a vertex hierarchy (Figure 3), and a selectively refined mesh corresponds to a "vertex front" through this hierarchy (e.g. M^0 and $\hat{M}$ in Figure 3). Our vertex hierarchy differs in two respects. First, vertices are renamed as they are split, and this

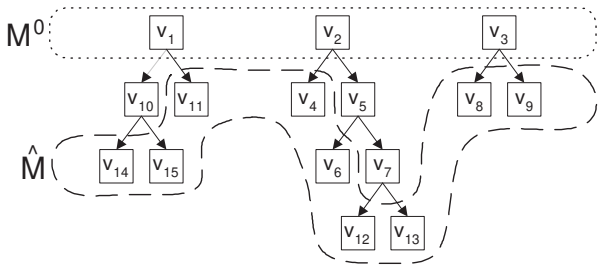


Figure 3: The vertex hierarchy on $\mathcal{V}$ forms a "forest", in which the root nodes are the vertices of the coarsest mesh (base mesh M^0) and the leaf nodes are the vertices of the most refined mesh (original mesh $\hat{M}$).

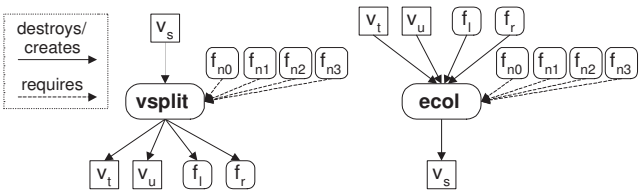


Figure 4: Preconditions and effects of *vsplit* and *ecol* transformations.

renaming contributes to the refinement dependencies. Second, the hierarchy is constructed top-down after loading a PM using a simple traversal of the *vsplit* records. Although our hierarchies may be unbalanced, they typically have fewer levels than in [24] (e.g. 24 instead of 65 for the bunny) because they are unconstrained.

Preconditions We define a set of preconditions for *vsplit* and *ecol* to be legal (refer to Figure 4).

A $vsplit(v_s, v_t, v_u, \ldots)$ transformation is legal if

(1) v_s is an active vertex, and

(2) the faces $\{f_{n0}, f_{n1}, f_{n2}, f_{n3}\}$ are all active faces.

An $ecol(v_s, v_t, v_u, \ldots)$ transformation is legal if

(1) v_t and v_u are both active vertices, and

(2) the faces adjacent to f_l and f_r are $\{f_{n0}, f_{n1}, f_{n2}, f_{n3}\}$, in the configuration of Figure 2.

Properties Let $\mathcal{M}^\star$ be the set of meshes obtained by transitive closure of legal *vsplit* and *ecol* transformations from M^0 (or equivalently from $\hat{M}$ since the PM sequence $M^0 \longleftrightarrow \hat{M}$ is legal). For any mesh $M = (V, F) \in \mathcal{M}^\star$, we observe the following properties:[1]

- If $vsplit(v_s, v_t, v_u, \ldots)$ is legal, then $\{f_{n0}, f_{n1}\}$ and $\{f_{n2}, f_{n3}\}$ must be pairwise adjacent and adjacent to v_s as in Figure 2.
- If the active vertex front lies below $ecol(v_s, v_t, v_u, \ldots)$ (i.e. $f_l, f_r \in F$), then $\{f_{n0}, f_{n1}, f_{n2}, f_{n3}\}$ must all be active.
- $M \in \mathcal{M}$, i.e. $M = M^S$ for some subsequence S, i.e. $\mathcal{M}^\star = \mathcal{M}$.
- $M = M^S$ is identical to the mesh obtained by applying to $\hat{M}$ the complement subsequence $\{n-1, \ldots, 0\} \setminus S$ of *ecol* transformations, which are legal.

Implementation To make these ideas more concrete, Figure 5 lists the C++ data structures used in our implementation. A selectively refinable mesh consists of an array of vertices and an array of faces. Of these vertices and faces, only a subset are active, as specified by two doubly-linked lists that thread through a subset of

[1] Although these properties have held for the numerous experiments we have performed, we unfortunately do not have formal proofs for them as yet.

```
struct ListNode {              // Node possibly on a linked list
    ListNode* next;            // 0 if this node is not on the list
    ListNode* prev;
};
struct Vertex {
    ListNode active;           // list stringing active vertices V
    Point point;
    Vector normal;
    Vertex* parent;            // 0 if this vertex is in M^0
    Vertex* vt;                // 0 if this vertex is in M̂; (vu=vt+1)
 // Remaining fields encode vsplit information, defined if vt ≠ 0.
    Face* fl;                  // (fr=fl+1)
    Face* fn[4];               // required neighbors fn0, fn1, fn2, fn3
    RefineInfo refine_info;    // defined in Section 4
};
struct Face {
    ListNode active;           // list stringing active faces F
    int matid;                 // material identifier
 // Remaining fields are used if the face is active.
    Vertex* vertices[3];       // ordered counter-clockwise
    Face* neighbors[3];        // neighbors[i] across from vertices[i]
};
struct SRMesh {                // Selectively refinable mesh
    Array<Vertex> vertices;    // set V of all vertices
    Array<Face> faces;         // set F̂ of all faces
    ListNode active_vertices;  // head of list V ⊆ V
    ListNode active_faces;     // head of list F ⊆ F̂
};
```

Figure 5: Principal C++ data structures.

the records. In the Vertex records, the fields *parent* and *vt* encode the vertex hierarchy of Figure 3. If a vertex can be split, its *fl* and *fn*[0..3] fields encode the remaining parameters of the *vsplit* (and hence the dependencies of Figure 4). Each Face record contains links to its current vertices, links to its current face neighbors, and a material identifier used for rendering.

4 REFINEMENT CRITERIA

In this section, we describe a query function qrefine(v_s) that determines whether a vertex v_s should be split based on the current view parameters. As outlined below, the function uses three criteria: the view frustum, surface orientation, and screen-space geometric error. Because qrefine is often evaluated thousands of times per frame, it has been designed to be fast, at the expense of a few simplifying approximations where noted.

```
function qrefine(v_s)
    // Refine only if it affects the surface within the view frustum.
    if outside_view_frustum(v_s) return false
    // Refine only if part of the affected surface faces the viewer.
    if oriented_away(v_s) return false
    // Refine only if screen-projected error exceeds tolerance τ.
    if screen_space_error(v_s) ≤ τ return false
    return true
```

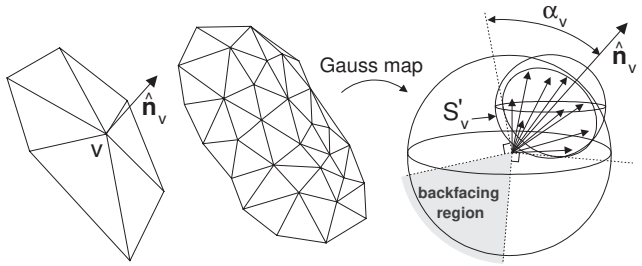
View frustum This first criterion seeks to coarsen the mesh outside the view frustum in order to reduce graphics load. Our approach is to compute for each vertex $v \in \mathcal{V}$ the radius r_v of a sphere centered at $\mathbf{v}$ that bounds the region of $\hat{M}$ supported by v and all its descendants. We let qrefine(v) return *false* if this bounding sphere lies completely outside the view frustum.

The radii r_v are computed after a PM representation is loaded into memory using a bounding sphere hierarchy as follows. First, we compute for each $v \in \hat{V}$ (the leaf nodes of the vertex hierarchy) a sphere S_v that bounds its adjacent vertices in $\hat{M}$. Next, we perform a postorder traversal of the vertex hierarchy (by scanning the *vsplit* sequence backwards) to assign each parent vertex v_s the smallest sphere S_{v_s} that bounds the spheres S_{v_t}, S_{v_u} of its two children. Finally, since the resulting spheres S_v are not centered on the vertices, we compute at each vertex v the radius r_v of a larger sphere centered at $\mathbf{v}$ that bounds S_v.

Since the view frustum is a 4-sided semi-infinite pyramid, a sphere of radius r_v centered at $\mathbf{v} = (v_x, v_y, v_z)$ lies outside the frustum if

$$a_i v_x + b_i v_y + c_i v_z + d_i < -r_v \quad \text{for any } i = 1 \ldots 4$$

where each linear functional $a_i x + b_i y + c_i z + d_i$ measures the signed Euclidean distance to a side of the frustum. Selective refinement based solely on the view frustum is demonstrated in Figure 12a.



(a) N_v (b) region of $\hat{M}$ (c) S^2

Figure 6: Illustration of (a) the neighborhood of v, (b) the region in $\hat{M}$ affected by v, and (c) the space of normals over that region and the cone of normals that bounds it.

Surface orientation The purpose of the second criterion is to coarsen regions of the mesh oriented away from the viewer, again to reduce graphics load. Our approach is analogous to the view frustum criterion, except that we now consider the space of normals over the surface (the Gauss map) instead of the surface itself. The space of normals is a subset of the unit sphere $S^2 = \{\mathbf{p} \in \mathbf{R}^3 : \|\mathbf{p}\| = 1\}$; for a triangle mesh $\hat{M}$, it consists of a discrete set of points, each corresponding to the normal of a triangle face of $\hat{M}$.

For each vertex v, we bound the space of normals associated with the region of $\hat{M}$ supported by v and its descendants, using a *cone of normals* [22] defined by a semiangle α_v about the vector $\hat{\mathbf{n}}_v$ = *v.normal* (Figure 6). The semiangles α_v are computed after a PM representation is loaded into memory using a *normal space hierarchy* [12]. As before, we first hierarchically compute at each vertex v a sphere S'_v that bounds the associated space of normals. Next, we compute at each vertex v the semiangle α_v of a cone about $\hat{\mathbf{n}}_v$ that bounds the intersection of S'_v and S^2. We let $\alpha_v = \frac{\pi}{2}$ if no bounding cone (with $\alpha_v < \frac{\pi}{2}$) exists.

Given a viewpoint $\mathbf{e}$, it is unnecessary to split v if $\mathbf{e}$ lies in the *backfacing region* of v, that is, if

$$\frac{\mathbf{a}_v - \mathbf{e}}{\|\mathbf{a}_v - \mathbf{e}\|} \cdot \hat{\mathbf{n}}_v > \sin \alpha_v \ ,$$

where $\mathbf{a}_v$ is a cone *anchor point* that takes into account the geometric bounding volume S_v (see [22] for details). However, to improve both space and time efficiency, we approximate $\mathbf{a}_v$ by $\mathbf{v}$ (it amounts to a parallel projection approximation [13]), and instead use the test

$$(\mathbf{v} - \mathbf{e}) \cdot \hat{\mathbf{n}}_v > 0 \quad \textbf{and} \quad ((\mathbf{v} - \mathbf{e}) \cdot \hat{\mathbf{n}}_v)^2 > \|\mathbf{v} - \mathbf{e}\|^2 \sin^2 \alpha_v \ .$$

The effect of this test is seen in Figures 13c, 14, and 16c, where the backfacing regions of the meshes are kept coarse.

Screen-space geometric error The goal of the third criterion is to adapt the mesh refinement such that the distance between the approximate surface M and the original $\hat{M}$, when projected on the screen, is everywhere less than a screen-space tolerance τ.

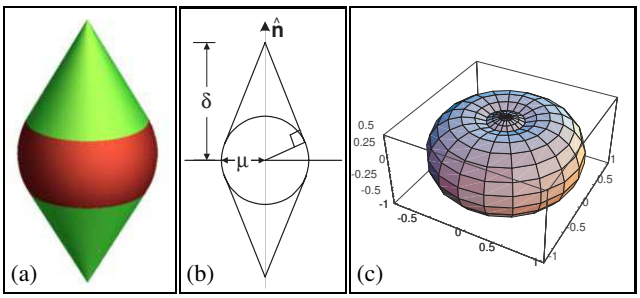


Figure 7: Illustration of (a) the deviation space $D_{\hat{\mathbf{n}}}(\mu, \delta)$, (b) its cross-section, and (c) the extent of its screen-space projection as a function of viewing angle (with $\mu = 0.5$ and $\delta = 1$).

To determine whether a vertex $v \in \mathcal{V}$ should be split, we seek a measure of the deviation between its current neighborhood N_v (the set of faces adjacent to v) and the corresponding region $\hat{N}_v$ in $\hat{M}$. One quantitative measure is the *Hausdorff distance* $\mathcal{H}(N_v, \hat{N}_v)$, defined as the smallest scalar r such that any point on N_v is within distance r of a point on $\hat{N}_v$, and vice versa. Mathematically, $\mathcal{H}(N_v, \hat{N}_v)$ is the smallest r for which $N_v \subset \hat{N}_v \oplus B(r)$ and $\hat{N}_v \subset N_v \oplus B(r)$ where $B(r)$ is the closed ball of radius r and $\oplus$ denotes the Minkowski sum[2]. If $\mathcal{H}(N_v, \hat{N}_v) = r$, the screen-space approximation error is bounded by the screen-space projection of the ball $B(r)$.

If N_v and $\hat{N}_v$ are similar and approximately planar, a tighter distance bound can be obtained by replacing the ball $B(r)$ in the above definition by a more general *deviation space* D. For instance, Lindstrom et al. [14] record deviation of height fields (graphs of functions over the $\mathbf{xy}$ plane) by associating to each vertex a scalar value δ representing a vertical deviation space $D_{\hat{\mathbf{z}}}(\delta) = \{h\,\hat{\mathbf{z}} \,:\, -\delta \le h \le \delta\}$. The main advantage of using $D_{\hat{\mathbf{z}}}(\delta)$ is that its screen-space projection vanishes as its principal axis $\hat{\mathbf{z}}$ becomes parallel to the viewing direction, unlike the corresponding $B(\delta)$.

To generalize these ideas to arbitrary surfaces, we define a deviation space $D_{\hat{\mathbf{n}}}(\mu, \delta)$ shown in Figure 7a–b. The motivation is that most of the deviation is orthogonal to the surface and is captured by a directional component $\delta\hat{\mathbf{n}}$, but a uniform component μ may be required when $\hat{N}_v$ is curved. The uniform component also allows accurate approximation of discontinuity curves (such as surface boundaries and material boundaries) whose deviations are often tangent to the surface. The particular definition of $D_{\hat{\mathbf{n}}}(\mu, \delta)$ corresponds to the shape whose projected radius along a direction $\vec{\mathbf{v}}$ has the simple formula $\max(\mu, \delta\|\hat{\mathbf{n}} \times \vec{\mathbf{v}}\|)$. As shown in Figure 7c, the graph of this radius as a function of view direction has the shape of a sphere of radius μ unioned with a "bialy" [14] of radius δ.

During the construction of a PM representation, we precompute μ_v, δ_v for deviation space $D_{\hat{\mathbf{n}}_v}(\mu_v, \delta_v)$ at each vertex $v \in \mathcal{V}$ as follows. After each $ecol(v_s, v_t, v_u, \ldots)$ transformation is applied, we estimate the deviation between N_{v_s} and $\hat{N}_{v_s}$ by examining the residual error vectors $E = \{\mathbf{e}_i\}$ from a dense set of points X sampled on $\hat{M}$ that locally project onto N_{v_s}, as explained in more detail in [10]. We use $\max_{\mathbf{e}_i \in E}(\mathbf{e}_i \cdot \hat{\mathbf{n}}_v) \,/\, \max_{\mathbf{e}_i \in E} \|\mathbf{e}_i \times \hat{\mathbf{n}}_v\|$ to fix the ratio δ_v/μ_v, and find the smallest $D_{\hat{\mathbf{n}}_v}(\mu_v, \delta_v)$ with that ratio that bounds E. Alternatively, other simplification schemes such as [2, 5, 9] could be adapted to obtain deviation spaces with guaranteed bounds.

Note that the computation of μ_v, δ_v does not measure parametric distortion. This is appropriate for texture-mapped surfaces if the texture is geometrically projected or "wrapped". If instead, vertices were to contain explicit texture coordinates, the residual computation could be altered to measure deviation parametrically.

Given viewpoint $\mathbf{e}$, screen-space tolerance τ (as a fraction of viewport size), and field-of-view angle φ, qrefine(v) returns *true* if the screen-space projection of $D_{\hat{\mathbf{n}}_v}(\mu_v, \delta_v)$ exceeds τ, that is, if

$$\max\left(\mu_v,\ \delta_v \left\| \hat{\mathbf{n}}_v \times \frac{\mathbf{v}-\mathbf{e}}{\|\mathbf{v}-\mathbf{e}\|} \right\|\right) / \|\mathbf{v}-\mathbf{e}\| \ \ge\ \left(2\cot\frac{\varphi}{2}\right)\tau\ .$$

For efficiency, we use the equivalent test

$$\begin{aligned} \mu_v^2 \ &\ge\ \kappa^2 \|\mathbf{v}-\mathbf{e}\|^2 \quad \textbf{or} \\ \delta_v^2\ \left(\|\mathbf{v}-\mathbf{e}\|^2 - ((\mathbf{v}-\mathbf{e})\cdot\hat{\mathbf{n}}_v)^2\right) \ &\ge\ \kappa^2 \|\mathbf{v}-\mathbf{e}\|^4\ , \end{aligned}$$

where $\kappa^2 = (2\cot\frac{\varphi}{2})^2\tau^2$ is computed once per frame. Note that the test reduces to that of [14] when $\mu_v = 0$ and $\hat{\mathbf{n}}_v = \hat{\mathbf{z}}$, and requires only a few more floating point operations in the general case. As seen in Figures 13b and 16b, our test naturally results in more refinement near the model silhouette where surface deviation is orthogonal to the view direction.

Our test provides only an approximate bound on the screen-space projected error, for a number of reasons. First, the test slightly underestimates error away from the viewport center, as pointed out in [14]. Second, a parallel projection assumption is made when projecting $D_{\hat{\mathbf{n}}}$ on the screen, as in [14]. Third, the neighborhood about v when evaluating qrefine(v) may be different from that in the PM sequence since M is selectively refined; thus the deviation spaces $D_{\hat{\mathbf{n}}}$ provide strict bounds only at the vertices themselves. Nonetheless, the criterion works well in practice, as demonstrated in Figures 12–16.

Implementation We store in each Vertex.RefineInfo record the four scalar values $\{-r_v, \sin^2\alpha_v, \mu_v^2, \delta_v^2\}$. Because the three refinement tests share several common subexpressions, evaluation of the complete qrefine function requires remarkably few CPU cycles on average (230 cycles per call as shown in Table 2).

5 INCREMENTAL SELECTIVE REFINEMENT ALGORITHM

We now present an algorithm for incrementally adapting a mesh within the selective refinement framework of Section 3, using the qrefine function of Section 4. The basic idea is to traverse the list of active vertices V before rendering each frame, and for each vertex $v \in V$, either leave it as is, split it, or collapse it. The core of the traversal algorithm is summarized below.

```
procedure adapt_refinement()
    for each v ∈ V
        if v.vt and qrefine(v)
            force_vsplit(v)
        else if v.parent and ecol_legal(v.parent) and
                not qrefine(v.parent)
            ecol(v.parent)      // (and reconsider some vertices)
procedure force_vsplit(v') {
    stack ← v'
    while v ← stack.top()
        if v.vt and v.fl ∈ F
            stack.pop()         // v was split earlier in the loop
        else if v ∉ V
            stack.push(v.parent)
        else if vsplit_legal(v)
            stack.pop()
            vsplit(v)           // (placing v.vt and v.vu next in list V)
        else for i ∈ {0...3}
            if v.fn[i] ∉ F
                // force vsplit that creates face v.fn[i]
                stack.push(v.fn[i].vertices[0].parent) [3]
```

[2] The Minkowski sum is simply $A \oplus B = \{\mathbf{a}+\mathbf{b} \,:\, \mathbf{a} \in A, \mathbf{b} \in B\}$.

[3] Implementation detail: the vertex that should be split to create an inactive face f is found in $f.vertices[0].parent$ because we always set both $f_l.vertices[0] = v_t$ and $f_r.vertices[0] = v_t$ when creating faces, thereby obviating the need for a Face.*parent* field.

We iterate through the doubly linked list of active vertices V. For any active vertex $v \notin \hat{M}$, if qrefine(v) evaluates to *true*, the vertex should be split. If *vsplit*(v) is not legal (i.e. if any of the faces $v.fn[0..3]$ are not active), a chain of other vertex splits are performed in order for *vsplit*(v) to become legal (procedure force_vsplit), namely those that introduce the faces $v.fn[0..3]$, and recursively, any others required to make those vertex splits legal.

For any active vertex $v \notin M^0$, if qrefine($v.parent$) returns *false*, the vertex v should be collapsed. However, this edge collapse is only performed if it is legal (i.e. if the sibling of v is also active and the neighboring faces of $v.parent.fl$ and $v.parent.fr$ match those of $v.parent.fn[0..3]$).

In short, the strategy is to force refinement when desired, but to coarsen only when possible. After a *vsplit* or *ecol* is performed, some vertices in the resulting neighborhood should be considered for further transformations. Since these vertices may have been previously visited in the traversal of V, we relocate them in the list to lie immediately after the list iterator. Specifically, following *vsplit*(v), we add $v.vt, v.vu$ after the iterator; and, following *ecol*($v.parent$), we add $v.parent$ and relocate v_l, v_r after the iterator (where v_l and v_r are the current neighbors of v as in Figure 1).

Time complexity The time complexity for adapt_refinement, transforming M^A into M^B, is $O(|V^A| + |V^B|)$ in the worst case since $M^A \rightarrow M^0 \rightarrow M^B$ could possibly require $O(|V^A|)$ *ecol*'s and $O(|V^B|)$ *vsplit*'s, each taking constant time. For continuous view changes, V^B is usually similar to V^A, and the simple traversal of the active vertex list is the bottleneck of the incremental refinement algorithm, as shown in Table 2. Note that the number $|V|$ of active vertices is typically much smaller than the number $|\hat{V}|$ of original vertices. The rendering process, which has the same time complexity ($|F| \simeq 2|V|$), in fact has a larger time constant. Indeed, adapt_refinement requires only about 14% of total frame time, as discussed in Section 8.

Regulation For a given PM and a constant screen-space tolerance τ, the number $|F|$ of active faces can vary dramatically depending on the view. Since both refinement times and rendering times are closely correlated to $|F|$, this leads to high variability in frame rates (Figure 9). We have implemented a simple scheme for regulating τ so as to maintain $|F|$ at a nearly constant level. Let m be the desired number of faces. Prior to calling adapt_refinement at time frame t, we set $\tau_t = \tau_{t-1}(|F_{t-1}|/m)$ where $|F_{t-1}|$ is the number of active faces in the previously drawn frame. As shown in Figure 10, this simple feedback control system exhibits good stability for our terrain flythrough. More sophisticated control strategies may be necessary for heterogeneous, irregular models. Direct regulation of frame rate could be attempted, but since frame rate is more sensitive to operating system "hiccups", it may be best achieved indirectly using a secondary, slower controller adjusting m.

Amortization Since the main loop of adapt_refinement is a simple traversal of the list V, we can distribute its work over consecutive frames by traversing only a fraction of V each frame. For slowly changing view parameters, this reduces the already low overhead of selective refinement while introducing few visual artifacts.

With amortization, however, regulation of $|F|$ through adjustment of τ becomes more difficult, since the response in $|F|$ may lag several frames. Our current strategy is to wait several frames until the entire list V has been traversed before making changes to τ. To reduce overshooting, we disallow *vsplit* refinement if the number of active faces reaches an upper limit (e.g. $|F| \geq 1.2m$). but do count the number of faces that would be introduced towards the next adjustment to τ. In the flythrough example of Figure 10, where the average frame rate is 7.2 frames/sec, amortization increases frame rate to 8 frames/sec.

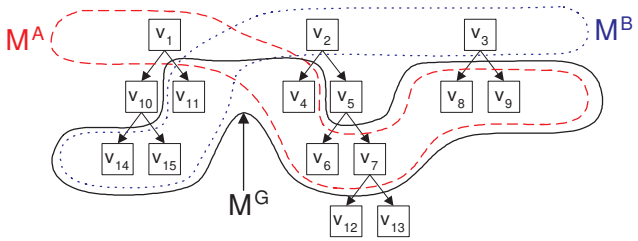


Figure 8: Illustration of two selectively refined meshes M^A and M^B, and of the mesh M^G used to geomorph between them.

Geomorphs The selective refinement framework also supports geomorphs between any two selectively refined meshes M^A and M^B. That is, one can construct a mesh $M^G(\alpha)$ whose vertices vary as a function of a parameter $0 \leq \alpha \leq 1$, such that $M^G(0)$ looks identical to M^A and $M^G(1)$ looks identical to M^B. The key is to first find a mesh M^G whose active vertex front is everywhere lower than or equal to that of M^A and M^B, as illustrated in Figure 8. Mesh $\hat{M}$ trivially satisfies this property, but a simpler mesh M^G is generally obtained by starting from either M^A or M^B and successively calling force_vsplit to advance the vertex front towards that of the other mesh. The mesh M^G has the property that its faces F^G are a superset of both F^A and F^B, and that any vertex $v_j \in V^G$ has a unique ancestor $v_{\rho^{G\rightarrow A}(j)} \in V^A$ and a unique ancestor $v_{\rho^{G\rightarrow B}(j)} \in V^B$. The geomorph $M^G(\alpha)$ is the mesh $(F^G, V^G(\alpha))$ with

$$\mathbf{v}_j^G(\alpha) = (1-\alpha)\mathbf{v}_{\rho^{G\rightarrow A}(j)} + (\alpha)\mathbf{v}_{\rho^{G\rightarrow B}(j)} \ .$$

In the case that M^B is the result of calling adapt_refinement on M^A, the mesh M^G can be obtained more directly. Instead of a single pass through V in adapt_refinement, we make two passes: a refinement pass $M^A \rightarrow M^G$ where only *vsplit* are considered, and a coarsening pass $M^G \rightarrow M^B$ where only *ecol* are considered. In each pass, we record the sequence of transformations performed, allowing us to backtrack through the inverse of the *ecol* sequence to recover the intermediate mesh M^G, and to construct the desired ancestry functions $\rho^{G\rightarrow A}$ and $\rho^{G\rightarrow B}$. Such a geomorph is demonstrated on the accompanying video. Because of view coherence, the number of vertices that require interpolation is generally smaller than the number of active vertices. More research is needed to determine the feasibility and usefulness of generating geomorphs at runtime.

6 RENDERING

Many graphics systems require triangle strip representations for optimal rendering performance [7]. Because the mesh connectivity in our incremental refinement scheme is dynamic, it is not possible to precompute triangle strips. We use a greedy algorithm to generate triangle strips at every frame, as shown in Figure 12e. Surprisingly, the algorithm produces strips of adequate length (on average, 10–15 faces per "generalized" triangle strip under IRIS GL, and about 4.2 faces per "sequential" triangle strip under OpenGL), and does so efficiently (Table 2).

The algorithm traverses the list of active faces F, and at any face not yet rendered, begins a new triangle strip. Then, iteratively, it renders the face, checks if any of its neighbor(s) has not yet been rendered, and if so continues the strip there. Only neighbors with the same material are considered, so as to reduce graphics state changes. To reduce fragmentation, we always favor continuing generalized triangle strips in a clockwise spiral (Figure 12e). When the strip reaches a dead end, traversal of the list F resumes. One bit of the Face.*matid* field is used as a boolean flag to record rendered faces; these bits are cleared using a quick second pass through F.

Recently, graphics libraries have begun to support interfaces for immediate-mode rendering of (V, F) mesh representations (e.g. Direct3D DrawIndexedPrimitive and OpenGL glArrayElementArrayEXT). Although not used in our current prototype, such interfaces may be ideal for rendering selectively refined meshes.

7 OPTIMIZING PM CONSTRUCTION FOR SELECTIVE REFINEMENT

The PM construction algorithm of [10] finds a sequence of *vsplit* refinement transformations optimized for accuracy, without regard to the shape of the resulting vertex hierarchy. We have experimented with introducing a small penalty function to the cost metric of [10] to favor balanced hierarchies in order to minimize unnecessary dependencies. The penalty for $ecol(v_t, v_u)$ is $c\,(n_{v_t}+n_{v_u})$ where n_v is the number of descendants of v (including itself) and c is a user-specified parameter. We find that a small value of c improves results slightly for some examples (i.e. reduces the number of faces for a given error tolerance τ), but that as c increases, the hierarchies become quadtree-like and the results worsen markedly (Figure 17). Our conclusion is that it is beneficial to introduce a small bias to favor balanced hierarchies in the absence of geometric preferences.

8 RESULTS

Timing results We constructed a PM representation of a Grand Canyon terrain mesh of 600^2 vertices (717,602 faces), and truncated this PM representation to 400,000 faces. This preprocessing requires several hours but is done off-line (Table 1). Loading this PM from disk and constructing the SRMesh requires less than a minute (most of it spent computing r_v and α_v). Figures 9 and 10 show measurements from a 3-minute real-time flythrough of the terrain without and with regulation, on an SGI Indigo2 Extreme (150MHz R4400 with 128MB of memory). The measurements show that the time spent in adapt_refinement is approximately 14% of total frame time. In the accompanying video, amortization is used to reduce this overhead to 8% of total frame time. For the flythrough of Figure 10, code profiling and system monitoring reveal the timing breakdown shown in Table 2. Note that triangle strip generation is efficient enough to keep CPU utilization below 100%; the graphics system is in fact the bottleneck. On another computer with the same CPU but with an Impact graphics system, the average frame rate increases from 7.2 to 14.0 frames/sec.

Space requirements Table 1 shows the disk space required to store the PM representations and associated deviation parameters; both are compressed using GNU gzip. Positions, normals, and deviation parameters are currently stored as floating point, and should be quantized to improve compression.

Since $|\mathcal{V}| \simeq 2|\hat{V}|$ and $|\hat{F}| \simeq 2|\hat{V}|$, memory requirement for SRMesh is $O(|\hat{V}|)$. The current implementation is not optimized for space, and requires about $224\,|\hat{V}|$ bytes. The memory footprint could be reduced as follows. Since only about half of all vertices $\mathcal{V}$ can be split, it would be best to store the split information ($fl, fn[0..3]$, *refine_info*) in a separate array of "Vsplit" records indexed by vt. If space is always allocated for 2 faces per *vsplit*, the Vertex.*fl* field can be deleted and instead computed from vt. Scalar values in the RefineInfo record can be quantized to 8 bits with an exponential map as in [14]. Coordinates of points and normals can be quantized to 16 bits. Material identifiers are unnecessary if the mesh has only one material. Overall, these changes would reduce memory requirements down to about $140\,|\hat{V}|$ bytes.

For the case of height fields, the memory requirement per vertex far exceeds that of regular grid schemes [14]. However, the fully detailed mesh $\hat{M}$ may have arbitrary connectivity, and may therefore be obtained by pre-simplifying a given grid representation, possibly

Table 1: Statistics for the various data sets.

Model	Fully detailed $\hat{M}$		Disk (MB)		Mem.	V hier.	Constr.
	$\lvert V\rvert$	$\lvert F\rvert$	PM	$\{\mu, \delta\}$	(MB)	height	(mins)
canyon_{200}	40,000	79,202	1.3	0.3	8.9	29	47
canyon_{400}	160,000	318,402	5.0	1.1	35.8	32	244
canyon_{600}	360,000	717,602	11.0	2.6	80.6	36	627
" trunc.	200,600	400,000	6.6	1.5	44.9	35	627
sphere	9,902	19,800	0.3	0.1	2.2	19	11
teapot trunc.	5,090	10,000	0.2	0.0	1.1	20	12
gameguy	21,412	42,712	0.8	0.2	4.8	26	30
bunny	34,835	69,473	1.2	0.2	7.8	24	51

Table 2: CPU utilization (on a 150MHz MIPS R4400).

	procedure	% of frame time	cycles/call
User	adapt_refinement	14 %	-
	(vsplit)	(0 %)	2200
	(ecol)	(1 %)	4000
	(qrefine)	(4 %)	230
	render (tstrip/face)	26 %	600
	GL library	19 %	-
System	OS + graphics	21 %	-
	CPU idle	20 %	-

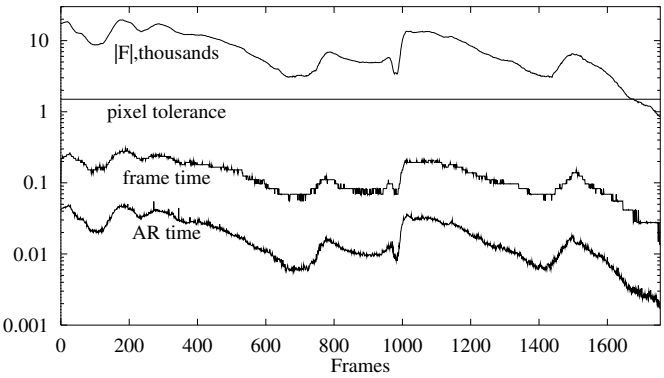


Figure 9: Measurements in flythrough for constant $\tau = 0.25\%$ (1.5 pixels in 600^2 window). From top: number of faces in thousands, τ in pixels, frame times and adapt_refinement times in seconds.

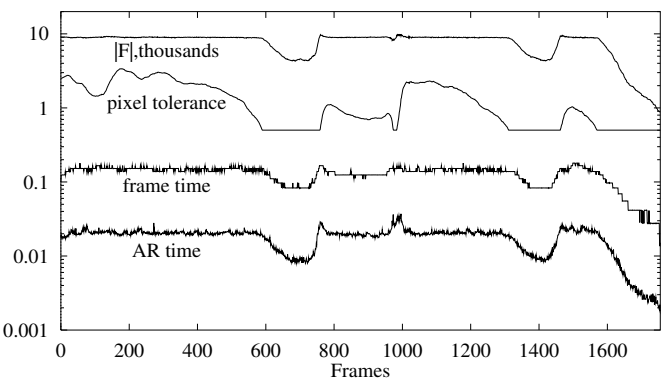


Figure 10: Same but with regulation to maintain $|F| \simeq 9000$. (τ is never allowed below 0.5 pixels.)

by an order of magnitude or more, without significant loss of accuracy. This pre-simplification may be achieved by simply truncating the PM representation, either at creation time or at load time.

Applications that use height fields often require efficient geometric queries, such as point search. Because the vertex hierarchies in our framework have $O(\log n)$ height in the average case (this can be enforced using the approach in Section 7), such queries can be performed in $O(\log n)$ time by iteratively calling force_vsplit on vertices in the neighborhood of the query point.

Parametric surfaces Our framework offers a novel approach to real-time adaptive tessellation of parametric surfaces. As a precomputation, we first obtain a dense tessellation of the surface, then construct from this dense mesh a PM representation, and finally truncate the PM sequence to a desired level of maximum accuracy. At runtime, we selectively refine this truncated PM representation according to the viewpoint (Figure 14). The main drawback of this approach is that the resolution of the most detailed tessellation is fixed a priori. However, the benefits include simplicity of runtime implementation (no trimming or stitching), efficiency (incremental, amortized work), and most importantly, high adaptability of the tessellations (accurate TIN's whose connectivities adapt both to surface curvature and to the viewpoint).

General meshes Figures 15 and 16 demonstrate selective refinement applied to general meshes. We expect this to be of practical use for rendering complex models and environments that do not conveniently admit scene hierarchies.

9 SUMMARY AND FUTURE WORK

We have introduced an efficient framework for selectively refining arbitrary progressive meshes, developed fast view-dependent refinement criteria, and presented an algorithm for incrementally adapting the approximating meshes according to these criteria. We have demonstrated real-time selective refinement on a number of meshes, including terrains, parametric surface tessellations, and general meshes. As the adaptive refinement algorithm exploits frame-to-frame coherence and is easily amortized, it consumes only a small fraction of total frame time. Because the selectively refined meshes stem from a geometrically optimized set of vertex split transformations with few dependencies, they quickly adapt to the underlying model, requiring fewer polygons for a given level of approximation than previous schemes.

There are a number of areas for future work, including:

- Memory management for large models, particularly terrains.
- Experimentation with runtime generation of geomorphs.
- Extension of refinement criteria to account for surface shading [24], or for surface velocity and proximity to gaze center [17].
- Adaptive refinement for animated models.
- Applications of selective refinement to collision detection.

ACKNOWLEDGMENTS

The Grand Canyon data is from the United States Geological Survey, with in-house processing by Chad McCabe of the Microsoft Geography Product Unit; the "gameguy" mesh is courtesy of Viewpoint DataLabs; the "bunny" is from the Stanford University Computer Graphics Laboratory. I also wish to thank Jed Lengyel, John Snyder, and Rick Szeliski for helpful comments, and Bobby Bodenheimer for useful discussions on control theory.

REFERENCES

[1] ABI-EZZI, S. S., AND SUBRAMANIAM, S. Fast dynamic tessellation of trimmed NURBS surfaces. *Computer Graphics Forum (Proceedings of Eurographics '94) 13*, 3 (1994), 107–126.

[2] BAJAJ, C., AND SCHIKORE, D. Error-bounded reduction of triangle meshes with multivariate data. *SPIE 2656* (1996), 34–45.

[3] CIGNONI, P., PUPPO, E., AND SCOPIGNO, R. Representation and visualization of terrain surfaces at variable resolution. In *Scientific Visualization '95* (1995), R. Scateni, Ed., World Scientific, pp. 50–68.

[4] CLARK, J. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19*, 10 (October 1976), 547–554.

[5] COHEN, J., VARSHNEY, A., MANOCHA, D., TURK, G., WEBER, H., AGARWAL, P., BROOKS, F., AND WRIGHT, W. Simplification envelopes. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), 119–128.

[6] DE FLORIANI, L., MARZANO, P., AND PUPPO, E. Multiresolution models for topographic surface description. *The Visual Computer 12*, 7 (1996), 317–345.

[7] EVANS, F., SKIENA, S., AND VARSHNEY, A. Optimizing triangle strips for fast rendering. In *Visualization '96 Proceedings* (1996), IEEE, pp. 319–326.

[8] FUNKHOUSER, T., AND SÉQUIN, C. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1993), 247–254.

[9] GUÉZIEC, A. Surface simplification with variable tolerance. In *Proceedings of the Second International Symposium on Medical Robotics and Computer Assisted Surgery* (November 1995), pp. 132–139.

[10] HOPPE, H. Progressive meshes. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), 99–108.

[11] KIRKPATRICK, D. Optimal search in planar subdivisions. *SIAM Journal on Computing 12*, 1 (February 1983), 28–35.

[12] KUMAR, S., AND MANOCHA, D. Hierarchical visibility culling for spline models. In *Proceedings of Graphics Interface '96* (1996), pp. 142–150.

[13] KUMAR, S., MANOCHA, D., AND LASTRA, A. Interactive display of large-scale NURBS models. In *1995 Symposium on Interactive 3D Graphics* (1995), ACM SIGGRAPH, pp. 51–58.

[14] LINDSTROM, P., KOLLER, D., RIBARSKY, W., HODGES, L., FAUST, N., AND TURNER, G. Real-time, continuous level of detail rendering of height fields. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), 109–118.

[15] LOUNSBERY, M., DEROSE, T., AND WARREN, J. Multiresolution surfaces of arbitrary topological type. *ACM Transactions on Graphics 16*, 1 (January 1997), 34–73.

[16] LUEBKE, D. Hierarchical structures for dynamic polygonal simplification. TR 96-006, Department of Computer Science, University of North Carolina at Chapel Hill, 1996.

[17] OHSHIMA, T., YAMAMOTO, H., AND TAMURA, H. Gaze-directed adaptive rendering for interacting with virtual space. In *Proc. of IEEE 1996 Virtual Reality Annual Intnl. Symp.* (1996), pp. 103–110.

[18] ROCKWOOD, A., HEATON, K., AND DAVIS, T. Real-time rendering of trimmed surfaces. In *Computer Graphics (SIGGRAPH '89 Proceedings)* (1989), vol. 23, pp. 107–116.

[19] ROSSIGNAC, J., AND BORREL, P. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics*, B. Falcidieno and T. L. Kunii, Eds. Springer-Verlag, 1993, pp. 455–465.

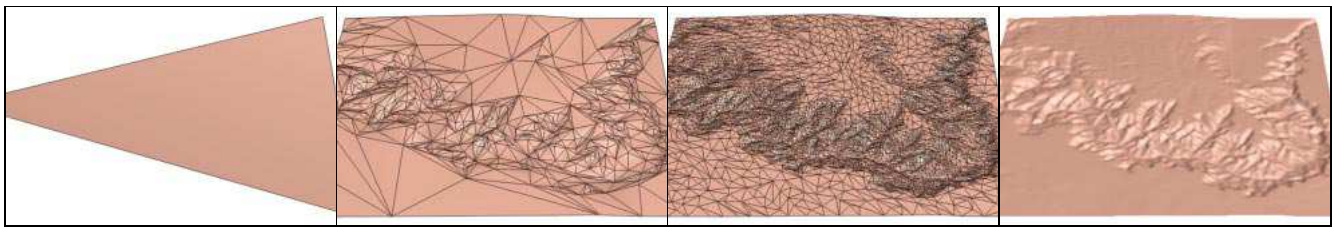
[20] SCARLATOS, L. L. A refined triangulation hierarchy for multiple levels of terrain detail. In *Proceedings, IMAGE V Conference* (June 1990), pp. 115–122.

[21] SCHROEDER, W., ZARGE, J., AND LORENSEN, W. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proceedings) 26*, 2 (1992), 65–70.

[22] SHIRMAN, L., AND ABI-EZZI, S. The cone of normals technique for fast processing of curved patches. *Computer Graphics Forum (Proceedings of Eurographics '93) 12*, 3 (1993), 261–272.

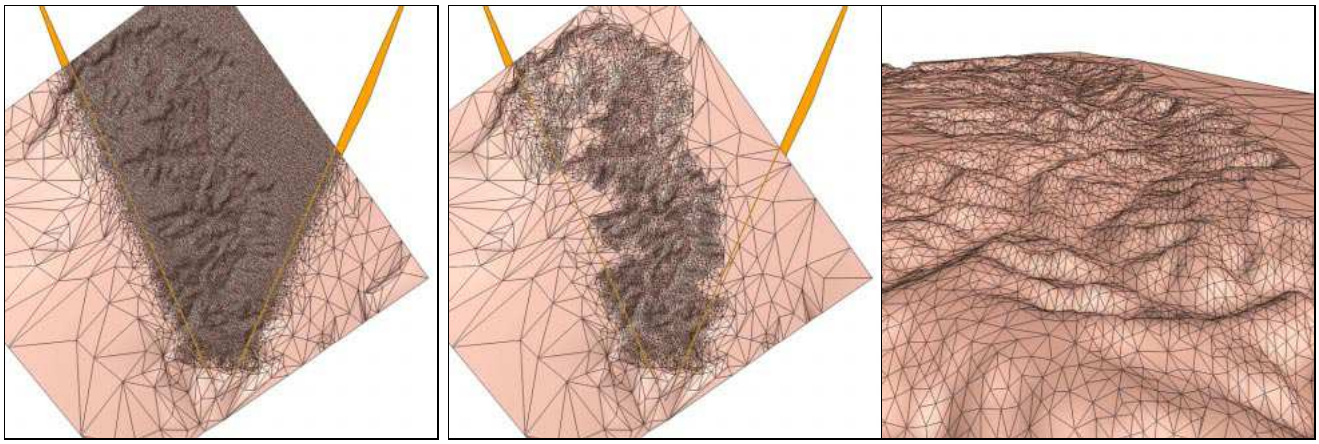
[23] TAYLOR, D. C., AND BARRETT, W. A. An algorithm for continuous resolution polygonalizations of a discrete surface. In *Proceedings of Graphics Interface '94* (1994), pp. 33–42.

[24] XIA, J., AND VARSHNEY, A. Dynamic view-dependent simplification for polygonal models. In *Visualization '96 Proceedings* (1996), IEEE, pp. 327–334.

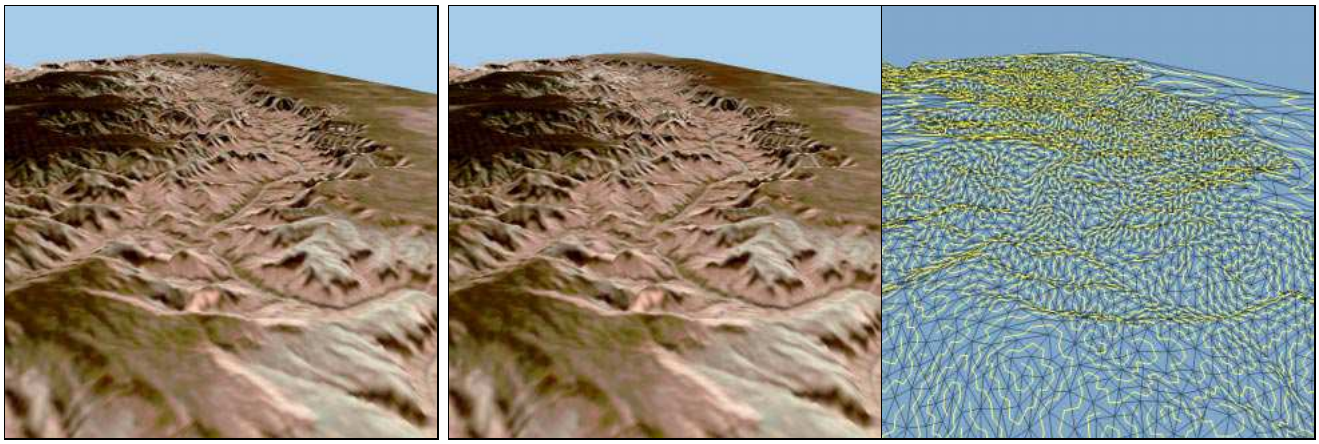


(a) Base mesh M^0 (1 face) (b) M^{514} (1,000 faces) (c) M^{5066} (10,000 faces) (d) $\hat{M} = M^n$ (79,202 faces)

Figure 11: The PM representation of a mesh $\hat{M}$ captures a continuous sequence of view-independent LOD meshes $M^0 \ldots M^n = \hat{M}$.

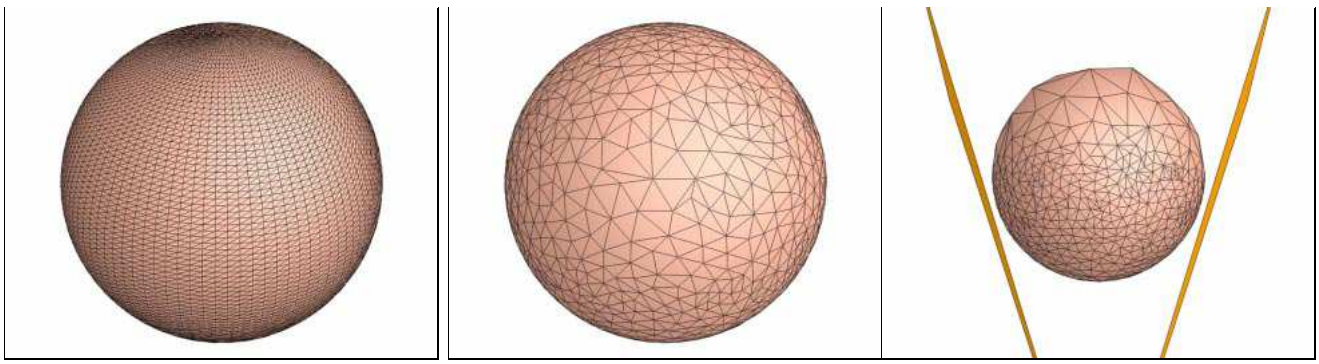


(a) Top view ($\tau = 0.0\%$; 33,119 faces) (b) Top and regular views ($\tau = 0.33\%$; 10,013 faces)



(c) Texture mapped $\hat{M}$ (79,202 faces) (d) Texture mapped (10,013 faces) (e) 764 generalized triangle strips

Figure 12: View-dependent refinement of the same PM, using the view frustum (highlighted in orange) and a screen-space geometric error tolerance of (a) 0% and (b,d,e) 0.33% of window size (i.e. 2 pixels for a 600×600 image).



(a) Original $\hat{M}$ (19,800 faces) (b) Front view and (c) Top view ($\tau = 0.075\%$; 1,422 faces)

Figure 13: View-dependent refinement of a tessellated sphere, demonstrating (b) the directionality of the deviation space $D_{\hat{\mathbf{n}}}$ (more refinement near silhouettes) and (c) the surface orientation criterion (coarsening of backfacing regions).

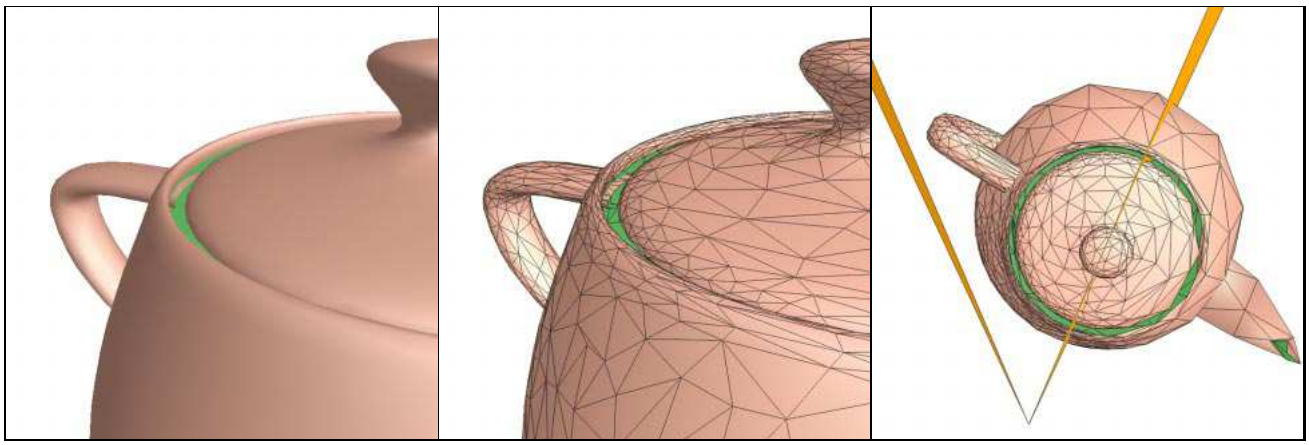
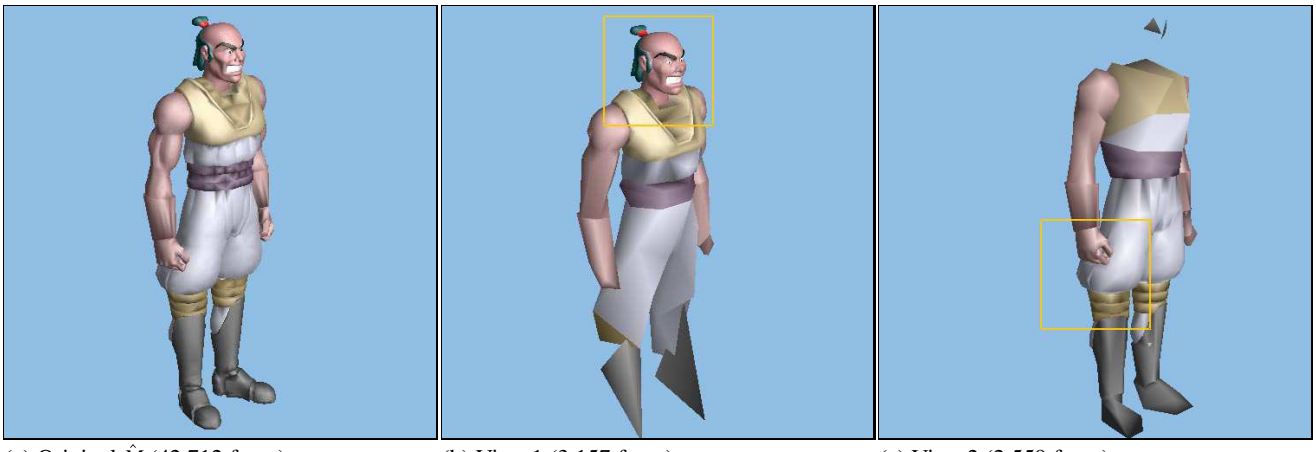
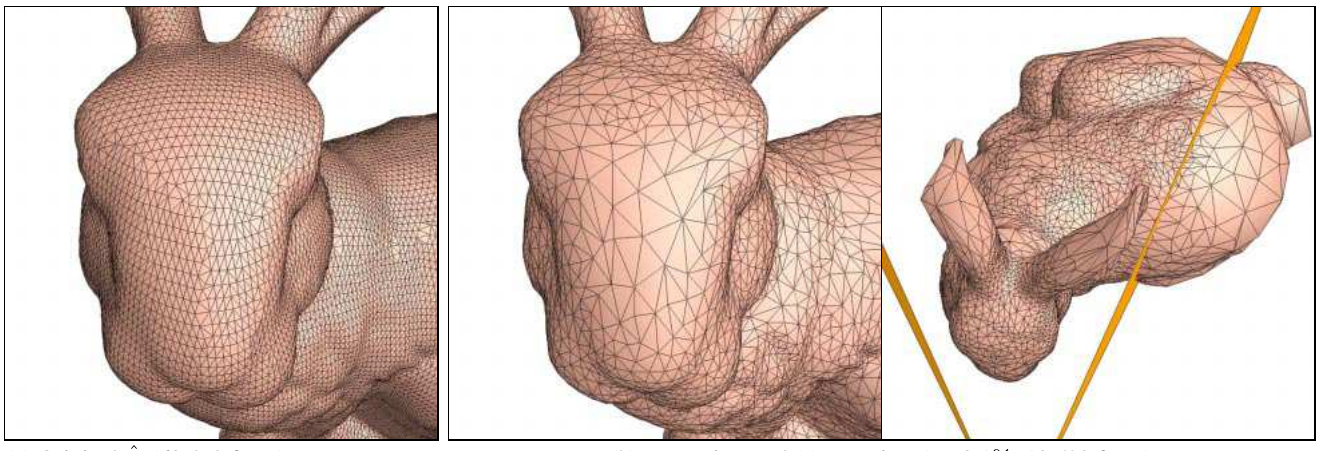


Figure 14: View-dependent refinement (τ = 0.15%; 1,782 faces) of a truncated PM representation (10,000 faces in $\hat{M}$) created from a tessellated parametric surface (25,440 faces). Interactive frame rate near this viewpoint is 14.7 frames/sec, versus 6.8 frames/sec using $\hat{M}$.



(a) Original $\hat{M}$ (42,712 faces) (b) View 1 (3,157 faces) (c) View 2 (2,559 faces)

Figure 15: Two view-dependent refinements of a general mesh $\hat{M}$ using view frustums highlighted in orange and with τ set to 0.6%.



(a) Original $\hat{M}$ (69,473 faces) (b) Front view and (c) Top view (τ = 0.1%; 10,528 faces)

Figure 16: View-dependent refinement. Interactive frame rate near this viewpoint is 6.7 frames/sec, versus 1.9 frames/sec using $\hat{M}$.

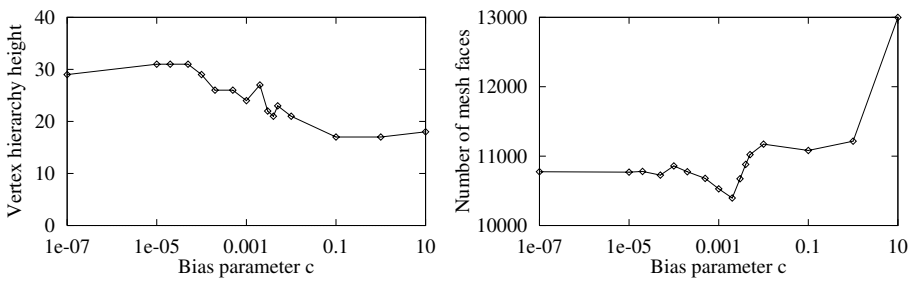


Figure 17: Height of vertex hierarchy, and number of faces in mesh of Figure 16b, as functions of the bias parameter c used in PM construction of bunny.