Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering

Hugues Hoppe
Microsoft Research

ABSTRACT

The key to real-time rendering of large-scale surfaces is to locally adapt surface geometric complexity to changing view parameters. Several schemes have been developed to address this problem of view-dependent level-of-detail control. Among these, the *view-dependent progressive mesh* (VDPM) framework represents an arbitrary triangle mesh as a hierarchy of geometrically optimized refinement transformations, from which accurate approximating meshes can be efficiently retrieved. In this paper we extend the general VDPM framework to provide temporal coherence through the runtime creation of *geomorphs*. These geomorphs eliminate "popping" artifacts by smoothly interpolating geometry. Their implementation requires new output-sensitive data structures, which have the added benefit of reducing memory use.

We specialize the VDPM framework to the important case of terrain rendering. To handle huge terrain grids, we introduce a block-based simplification scheme that constructs a progressive mesh as a hierarchy of block refinements. We demonstrate the need for an accurate approximation metric during simplification. Our contributions are highlighted in a real-time flyover of a large, rugged terrain. Notably, the use of geomorphs results in visually smooth rendering even at 72 frames/sec on a graphics workstation.

1 INTRODUCTION

Real-time visualization of large-scale surfaces is a challenging problem. As an example, Figure 1 shows a grid mesh of 4,097 × 2,049 vertices containing both color and elevation data. The most common approach for rendering such surfaces is to exploit the traditional 3D graphics pipeline, which is optimized to transform and texture-map triangles. The graphics pipeline has two main stages: *geometry processing* and *rasterization*.

Typically, the rasterization effort is relatively steady because the rendered surface has low depth complexity. In the worst case, the model covers the viewport, and the number of filled pixels is only slightly more than that in the frame buffer. Current graphics workstations (and soon, personal computers) have sufficient fill rate to texture-map the entire frame buffer at 30–72 Hz, even with advanced features like trilinear mip-map filtering and detail textures.

Instead, geometry processing proves to be the bottleneck. Even high-end platforms can process in real-time only a tiny fraction of the nearly 17 million triangles shown in Figure 1. Of course, there is little point in rendering more triangles than there are pixels. In fact, the surface usually exhibits significant spatial coherence, so that its perspective projection can be approximated to an accuracy of a few pixels by a much simpler mesh (e.g. 2,000-20,000 triangle faces) as demonstrated in Figure 11. Finding such a mesh, and updating it as the viewing parameters change, is referred to as *view-dependent level-of-detail (LOD) control*. The challenge is to locally adjust the complexity of the approximating mesh to satisfy a screen-space pixel tolerance while maintaining a rendered surface that is both spatially and temporally continuous. To be spatially continuous, the mesh should be free of cracks and T-junctions. To be temporally continuous, the rendered mesh should not visibly "pop" from one frame to the next.

Several schemes have been developed to address view-dependent LOD control, as summarized in Section 2. Among these, the *view-dependent progressive mesh* (VDPM) framework [16] represents an arbitrary triangle mesh as a hierarchy of geometrically optimized refinement transformations. Consequently, it is able to satisfy a given screen-space approximation tolerance with a simpler mesh — a key advantage in reducing the geometry bottleneck.

In this paper we first extend the general VDPM framework in the following two areas.

Memory requirements : We redesign the data structures to be output-sensitive, thereby reducing memory requirements (Section 4.1).

Runtime geomorphs : We introduce a scheme for efficient runtime creation of *geomorphs*, which smoothly transition surface geometry over several frames to eliminate popping (Section 4.2). To our knowledge this is the first runtime scheme for temporally smooth, view-dependent LOD control on arbitrary meshes.

Many types of graphics scenes have complex geometric descriptions. It should be emphasized, however, that scenes often contain many distinct small-scale objects, for which LOD can be adjusted independently [10] using traditional *view-independent* simplification techniques (e.g. [1, 3, 12, 15, 21]). In our opinion, the overhead of view-dependent LOD is only justified when necessary — for large-scale continuous surfaces. In outdoor scenes, the primary instance is the terrain surface. In other domains, examples include the virtual flythroughs of organic structures and of CAD surfaces like ship hulls [19, 23]. In this paper we place the emphasis on terrain rendering because of its importance in the growing entertainment market.

In the second half of the paper, we specialize the VDPM framework to the special case of terrains (i.e. height fields). In particular, we add the following two enhancements:

Approximation error : We demonstrate that the common approach of measuring approximation error on height fields solely at grid points is inadequate for view-dependent LOD using irregular meshes (Section 5.1). Fortunately, exact approximation error can be computed efficiently during a preprocessing step.

Scalability : To handle huge terrain models, we present a block-based recursive simplification process (Section 5.2). The result of this process is a hierarchical progressive mesh representation that permits runtime memory management.

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

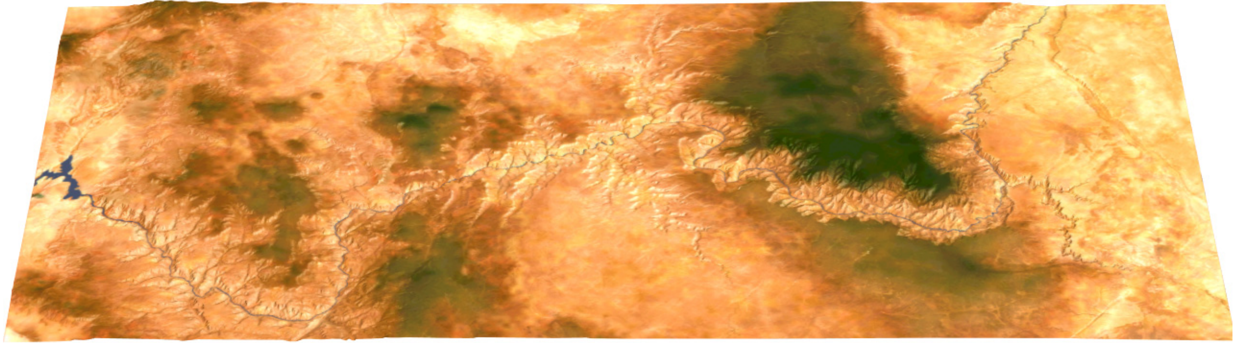


Figure 1: A terrain grid of 4,097 × 2,049 vertices containing both color and elevation data.

2 RELATED WORK

Height fields. Although there exist numerous multiresolution representations for height fields (see surveys in [7, 14]), only a subset support view-dependent LOD, and it is only recently that efficient on-line algorithms have been introduced that incrementally adapt LOD as the view parameters change.

Taylor and Barrett [22] extract mesh approximations from rectangular **quadtree** hierarchies. Both Lindstrom et al. [18] and Duchaineau et al. [8] define **bintree** hierarchies, based on binary subdivision of right isosceles triangles, and demonstrate real-time view-dependent LOD. Because these representations are based on regular subdivision, they offer concise storage. Duchaineau et al. are able to create optimal approximating meshes through incremental changes at each frame. However, the meshes are only optimal within a restricted space of meshes, since the regular subdivision structure constrains both vertex locations and face connectivities. As a result, the approximations may be far from optimal when one considers the space of all possible triangulations of the domain.

Several methods use **Delaunay triangulation** to develop multiresolution hierarchies [2, 5]. In particular, Cohen-Or and Levanoni [5] support on-line view-dependent LOD with temporal coherence, but must resort to "two-stage" geomorphs. Compared to quadtrees and bintrees, these methods allow more general distribution of vertices over the domain. However, the mesh connectivities are again constrained, in this case by the Delaunay triangulation criterion.

Arbitrary meshes. Xia and Varshney [23] and Hoppe [16] show that multiresolution hierarchies for arbitrary meshes can be defined using a general refinement transformation called a *vertex split* (Figure 2). Whereas Xia and Varshney construct the hierarchy using an edge length heuristic, Hoppe constructs it from the geometrically optimized sequence in a progressive mesh representation (Section 3). De Floriani et al. [6] introduce another related refinement hierarchy.

When applied to the special case of height fields, these frameworks are able to satisfy a given approximation error using fewer faces due to the absence of connectivity constraints. Lilleskog [17] reports that the VDPM scheme uses 50-75% of the number of active triangles required by bintree schemes for the same screen-space error.

3 REVIEW OF VIEW-DEPENDENT PROGRESSIVE MESH FRAMEWORK

As introduced in [15], a *progressive mesh* (PM) representation describes an arbitrary triangle mesh M^n as a coarse base mesh M^0 together with a sequence of n refinement transformations $\{vsplit_0, \ldots, vsplit_{n-1}\}$ called *vertex splits* (Figure 2) that progressively recover detail. A PM representation for M^n is obtained by carefully simplifying it using n successive *edge collapse* transformations (Figure 2), and recording their inverses.

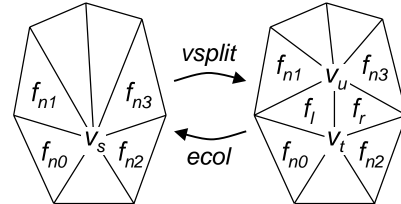


Figure 2: The *vertex split* refinement operation, and its inverse, the *edge collapse* coarsening operation.

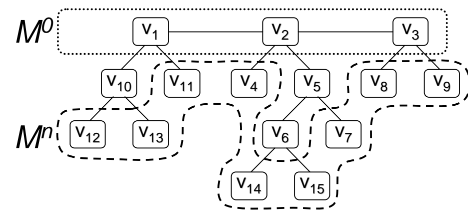


Figure 3: The *vsplit* refinement transformations uniquely define a vertex hierarchy.

As shown in the VDPM framework [16], this same sequence of *vsplit* refinement transformations uniquely defines a vertex hierarchy (Figure 3), in which the root nodes correspond to the vertices of the base mesh M^0, and the leaf nodes correspond to the fully detailed mesh M^n. This hierarchy permits the creation of selectively refined meshes, that is, meshes not necessarily in the original sequence $M^0 \ldots M^n$. A selectively refined mesh M corresponds to a "vertex front" through the vertex hierarchy (e.g. M^0 and M^n in Figure 3), and is obtained by incrementally applying *ecol* and *vsplit* transformations subject to a set of legality conditions. The selectively refined mesh M, also called the *active mesh*, is usually much simpler than the fully detailed mesh M^n.

To achieve view-dependent LOD, the active vertex front is traversed prior to rendering each frame, and each vertex may be either coarsened or refined based on view-dependent refinement criteria. In [16], a *vsplit* refinement is performed if its neighborhood satisfies 3 criteria: (1) it intersects the view frustum, (2) its Gauss map is not strictly oriented away, and (3) its screen-projected deviation from M^n exceeds a user-specified pixel tolerance τ. For efficient and conservative runtime evaluation of these criteria, each vertex in the hierarchy stores the following: a bounding-sphere radius r_v, a normal vector $\hat{\mathbf{n}}_v$, a cone-of-normals angle α_v, and a deviation space encoded by a uniform component μ_v and a directional component δ_v [16]. Geomorphs are demonstrated to be feasible within the VDPM framework, but their runtime creation is left as future work.

In the remainder of the paper, we assume that the size of the base mesh M^0 is insignificant compared to that of the fully refined M^n, and therefore assume that M^n has approximately n vertices and $2n$ faces. We let m denote the number of vertices in the active mesh M, so that M has approximately $2m$ faces. Typically, $m \ll n$.

4 EFFICIENT, TEMPORALLY SMOOTH VDPM

4.1 Output-sensitive data structures

One limitation of the original VDPM scheme [16] is that all its data structures scale proportionally with the size n of the fully refined mesh M^n. In particular, static storage is allocated to represent the mesh connectivity for all faces in M^n even though only a small fraction are usually active at any one time. To allow the introduction of geomorphs without prohibitive memory use, we have redesigned the data structures to be output-sensitive. As shown in Figure 4, the structures are separated into two parts: a static part encoding the vertex hierarchy and refinement dependencies (size $88n$ bytes), and a dynamic part encoding the connectivity of just the active mesh M (size $112m$ bytes).

Let us examine the data structures more closely. Each Vertex contains a pointer to its parent, and an index i of the $vsplit_i$ that creates its children (or -1 if it is a leaf). Since vertices are numbered consecutively, the index i is sufficient to compute the indices of the two child vertices v_t and v_u, and of the one/two child faces f_l and f_r. We make several optimizations to further reduce memory. Geometry storage is reduced in half by modifying the *vsplit*/*ecol* transformation to force vertices v_s and v_t to have the same geometry, as illustrated in Figure 2. Experiments reveal that this $\mathbf{v}_s = \mathbf{v}_t$ constraint results in an average increase of about 15% in active faces. Instead of storing the texture identifiers for the new faces f_l and f_r in Vsplit, we infer them during a *vsplit* from the adjacent active faces f_{n1} and f_{n3} respectively.

For concreteness, here is pseudocode for the vertex split transformation; procedure ecol(v_s) is defined analogously.

```
procedure vsplit(v_s)
    v_t ← &vertices[|V^0| + v_s.i * 2]
    v_u ← v_t + 1
    f_l ← &faces[|F^0| + v_s.i * 2]
    f_r ← f_l + 1
    f_n0..3 ← vsplits[v_s.i].fn[0..3]
    v_t.avertex ← v_s.avertex;   v_t.avertex.vertex ← v_t
    v_s.avertex ← 0
    v_u.avertex ← new AVertex;   v_u.avertex.vertex ← v_u
    v_u.avertex.listnode.add_to_list(active_vertices)
    v_u.avertex.vgeom ← vsplits[v_s.i].vu_vgeom
    f_l.aface ← new AFace;   f_l.aface.listnode.add_to(active_faces)
    [ Fill in entries of f_l.aface ]
    f_r.aface ← new AFace;   f_r.aface.listnode.add_to(active_faces)
    [ Fill in entries of f_r.aface ]
    [ Update f_n0..3.neighbors[..] to point to f_l, f_r ]
    [ For each face f around v_u: update f.vertices[..] : v_s → v_u ]
```

To enable the geomorphs described in Section 4.2, each active vertex has a field *vmorph*, which points to a dynamically allocated VertexMorph record when the vertex is morphing. In practice, the number g of morphing vertices is only a fraction of the number m of active vertices, which is itself only a small fraction of the total number n of vertices.

Overall, the new data structures need $88n + 112m + 52g$ bytes, compared to $224n$ bytes in [16].

Because in practice the number of active faces is $2m \approx 12{,}000 < 65{,}536$, the AVertex* and AFace* pointers in the static structure can be replaced by 16-bit indices. Additionally, in Vsplit we can quantize the coordinates to 16 bits and use 256-entry lookup tables for $\{r_v, \mu_v, \delta_v, \sin^2 \alpha_v\}$. Static storage is then reduced from $88n$ to $56n$ bytes. By way of comparison, a standard representation for a pre-simplified, quantized, irregular mesh uses $42n$ bytes of memory ($(n)(12)$ bytes for positions and normals, and $(2n)(3)(4)$ bytes for connectivity). Thus the VDPM framework only requires a 33% increase in memory over a static, non-LOD representation.

```
// Statically allocated structures. (space O(n))
struct VGeom                        // Vertex geometry
    Point point                     // position v
  † Vector normal                   // normal n̂_v
struct Vertex                       // Static vertex [2n]
    AVertex* avertex                // active vertex, 0 if inactive
    Vertex* parent                  // parent vertex, 0 if root
    int i                           // index of vsplit_i, -1 if leaf
struct Face                         // Static face [2n]
    AFace* aface                    // active face, 0 if inactive
struct Vsplit                       // Vertex split [n]
    VGeom vu_vgeom                  // geometry for child vertex v_u
    Face* fn[4]                     // required neighbors f_n0, f_n1, f_n2, f_n3
    float radius                    // max extent r_v of affected region
  † float sin2alpha                 // cone-of-normals angle (sin^2 α_v)
  † float uni_error                 // uniform error μ_v
    float dir_error                 // directional error δ_v
struct ListNode                     // Node on a doubly linked list
    ListNode* next
    ListNode* prev
struct SRMesh                       // Selectively refinable mesh
    Array<Vertex> vertices          // all vertices in hierarchy [2n]
    Array<Face> faces               // all faces [2n]
    Array<Vsplit> vsplits           // vertex splits vsplit_i [n]
    ListNode active_vertices        // head of active vertex list
    ListNode active_faces           // head of active face list

// Dynamically allocated structures. (space O(m))
struct AVertex                      // Active vertex (on heap) [m]
    ListNode listnode               // list stringing active vertices
    Vertex* vertex                  // pointer back to static vertex
    VGeom vgeom                     // vertex coordinates (x,y,z)
    VertexMorph* vmorph             // ≠ 0 if geomorphing (Section 4.2)
struct AFace                        // Active face (on heap) [2m]
    ListNode listnode               // list stringing active faces
    AVertex* vertices[3]            // ordered counter-clockwise
    AFace* neighbors[3]             // neighbors[j] across from vertices[j]
    int texture_id                  // texture tile identifier
struct VertexMorph                  // (on heap, see Section 4.2) [g]
    bool coarsening                 // true if coarsening, false if refining
    short gtime                     // # of geomorph frames remaining
    VGeom vg_refined                // refined geometry (back-up copy)
    VGeom vginc                     // increment per frame during morph
```

Figure 4: Principal C++ data structures. Fields denoted by '†' are omitted for terrain rendering (Section 5).

4.2 Runtime generation of geomorphs

Two factors are crucial to a good visual flythrough simulation: a (high) steady frame rate, and the absence of popping artifacts. At first, these two goals seem contradictory. Popping is avoided if the screen-space error tolerance is kept near a value of 1 pixel, but with a constant error tolerance, the number of active faces can vary greatly depending on the model complexity near the viewpoint, leading to non-uniform frame rate.

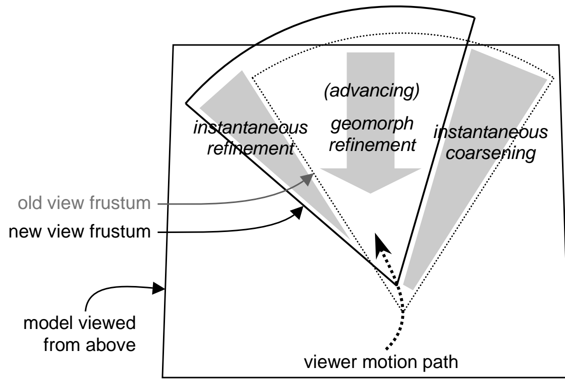


Figure 5: Changes to active mesh during forward motion of viewer.

Our solution is to aim primarily for a constant frame rate by adjusting the screen-space error tolerance, and to eliminate popping by smoothly morphing the geometry. Although the model may at times have a projected geometric error of a few pixels, results indicate that geomorphs make this error nearly imperceptible. The remainder of this section describes a scheme for generating geomorphs at runtime within the VDPM framework. The geomorph scheme is effective enough that we can increase the pixel error tolerance to improve frame rate (up to 72 frames/sec) with few noticeable artifacts.

The main idea is as follows. When the refinement criteria indicate the need for an *ecol* or *vsplit*, instead of performing the transformation instantaneously, we perform it as a geomorph by gradually changing the vertex geometry over several frames. Specifically, a transformation is performed as a geomorph if and only if the region of the affected surface is visible (i.e. the region overlaps with the view frustum and is not oriented away from the viewer). Indeed, it is undesirable to initiate a geomorph on a region known to be invisible, because according to the refinement criteria, such a region may have unbounded screen-space error. If such a region were to become visible prior to the end of the geomorph, it could lead to an arbitrarily large screen-space displacement. For example, as the viewpoint pans left, the nearby off-screen region should not be morphing from its coarse state as it enters the left edge of the viewport.

Besides position, other vertex attributes interpolated during a geomorph may include normal, color, and texture coordinates. Most attributes are linearly interpolated. Normals are interpolated over the unit sphere. In our examples, texture coordinates are generated implicitly during rendering using a linear map on vertex positions. Because the map is linear, these texture coordinates are identical to those that would result if texture coordinates were tracked explicitly at vertices.

Figure 5 illustrates the types of changes applied to the active mesh as a user moves forward and to the left through a model. Regions of the model entering the view frustum (on the left) are instantaneously refined. Regions leaving the view frustum (on the right and near the viewer) are instantaneously coarsened. Finally, regions within the view frustum are geomorph refined. Note that in this common case of forward viewer motion, geomorph coarsening does not occur.

Geomorph refinement. We first present geomorph refinements, as it is the more common case in practice. Only minor changes are made to the vsplit procedure described in Section 4.1. The mesh connectivity is still modified immediately, but we initially assign the new vertex v_u the same geometry as its sibling v_t, and only gradually modify the geometry of v_u to its refined state over the next *gtime* frames. The parameter *gtime* is user-specified; in our prototype we have set it equal to the frame rate (30–72), so that

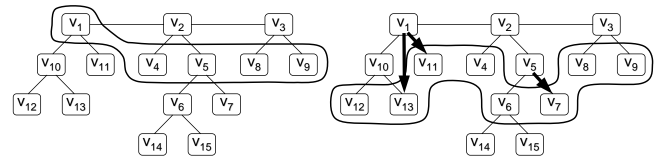


Figure 6: Illustration of geomorph refinement. The active mesh on the right is obtained by applying 3 vertex splits to the active mesh on the left. To obtain a smooth transition, the geometry for vertices $\{v_{13}, v_{11}, v_7\}$ are gradually interpolated from those of their ancestors as indicated by the arrows. (By construction, positions $\mathbf{v}_{12} = \mathbf{v}_1$ and $\mathbf{v}_6 = \mathbf{v}_5$, so no interpolation is necessary for them.)

geomorphs have a lifetime of one second. Note that the geomorphs do not require the introduction of additional faces, as the mesh connectivity is exactly that of the desired refined mesh. Here is the modified pseudocode:

```
function is_invisible(v_s)
  return outside_view_frustum(v_s) or
         oriented_away(v_s)   // see [16] for their definitions

procedure vsplit(v_s)
  ...    // code from Section 4.1
  if not is_invisible(v_s)
    v_u.avertex.vgeom ← v_t.avertex.vgeom
    vm ← v_u.avertex.vmorph ← new VertexMorph
    vm.coarsening ← false
    vm.gtime ← gtime
    vm.vg_refined ← vsplits[v_s.i].vu_vgeom
    vm.vginc ← (vm.vg_refined − v_u.avertex.vgeom)/gtime
```

It should be emphasized that this modified *vsplit* may be applied to a vertex v_s *already* morphing. In other words, geomorph refinements can be composed arbitrarily, even with overlapping lifetimes. Procedure vsplit simply advances the vertex front down the vertex hierarchy (possibly several "layers"), modifying the mesh connectivity instantaneously while deferring geometric changes. Figure 6 shows an example in which 3 vertex splits are performed.

As shown in the pseudocode below, at each frame we traverse the set of active vertices, and for each morphing vertex, we advance its geometry and decrement its *gtime* field. When *gtime* reaches 0, the vertex has reached its goal geometry and the VertexMorph record is deleted.

```
procedure update_vmorphs()
  for each v ∈ active_vertices
    if v.vmorph
      v.vgeom ← v.vgeom + v.vmorph.vginc
      v.vmorph.gtime ← v.vmorph.gtime − 1
      if v.morph.gtime = 0  delete v.vmorph
```

Geomorph coarsening. Geomorph coarsening is more challenging within the VDPM framework. Unlike in geomorph refinement, the geometry interpolation must take place first, and only then can the mesh connectivity be coarsened. Because the mesh must remain refined during the geomorph's lifetime, evaluating the legality of further coarsening steps is non-trivial. Moreover, even if this legality could be determined, further coarsening would in general require modifying several ongoing geomorphs. These difficulties are not inherent to the VDPM framework but should arise even in multiresolution hierarchies based on uniform subdivision.

As a consequence, we only allow geomorph coarsening "one layer at a time". That is, out of the set of desired geomorph coarsenings, we simultaneously perform all the currently legal ones, and make their dependents wait for these initial geomorphs to complete. To help mitigate this delay, we reduce the *gtime* parameter for geomorph coarsening to half that for geomorph refinement. Fortunately,

geomorph coarsening is required only when the viewer is moving backwards — a more infrequent situation.

Implementation of geomorph coarsening primarily involves changes to the function that adjusts the active vertex front:

```
procedure adapt_refinement()    // compare with definition in [16]
  for each v ∈ active_vertices
    vs ← v.vertex
    if vs.i ≥ 0 and not is_invisible(vs) and screen_error(vs) > τ
      force_vsplit(vs)
    else if vs.parent and ecol_legal(vs.parent)
      vmc ← (v.vmorph and v.vmorph.coarsening)
      if is_invisible(vs.parent)
        if vmc  finish_geomorph_coarsening(v)
        ecol(vs.parent)
      else if screen_error(vs.parent) > τ
        if vmc  abort_geomorph_coarsening(v)
      else if vmc
        if v.vmorph.gtime = 1
          finish_geomorph_coarsening(v)
          ecol(vs.parent)
      else
        start_geomorph_coarsening(vs)
```

Modified screen-space error metric. Recall that a geomorph refinement is initiated when the screen-projected deviation of its mesh neighborhood exceeds a pixel tolerance τ. If the viewer is moving forward, the mesh neighborhood is likely closer to the viewer by the time the geomorph completes, thus invalidating the error estimate. Our solution is to anticipate the viewer location *gtime* frames into the future when evaluating the screen-space error metric. We estimate this future location by extrapolation based on the current per-frame viewer velocity $\Delta\mathbf{e}$. A more rigorous solution to account for changes in velocity would require altering the lifetimes of ongoing geomorphs, which seems expensive.

The original refinement criterion from [16, 18] is:

$$\frac{\delta_v}{\|\mathbf{v}-\mathbf{e}\|}\sqrt{1-\left(\frac{(\mathbf{v}-\mathbf{e})\cdot\hat{\mathbf{n}}_v}{\|\mathbf{v}-\mathbf{e}\|}\right)^2} > \kappa\ ,$$

where $\mathbf{e}$ is the viewpoint, $\mathbf{v}$ the mesh vertex, $\hat{\mathbf{n}}_v$ its normal, δ_v its neighborhood's residual error, and $\kappa = 2\tau\tan\frac{\varphi}{2}$ accounts for field-of-view angle φ and pixel tolerance τ. The square-root factor allows greater simplification when the surface is viewed along the direction of its normal. For our terrain flyover, with fixed τ, this factor reduces the average number of active faces by only 3%, so we decided to omit it. The denominator $\|\mathbf{v}-\mathbf{e}\|$ is an estimate of the z coordinate of the vertex $\mathbf{v}$ in screen space. We replace this denominator with the linear functional $L_{\mathbf{e},\vec{\mathbf{e}}}(\mathbf{v}) = (\mathbf{v}-\mathbf{e})\cdot\vec{\mathbf{e}}$ which computes this z coordinate directly ($\vec{\mathbf{e}}$ is the viewing direction).

Our new screen-space error criterion is $\delta_v > \kappa\, L_{\mathbf{e}',\vec{\mathbf{e}}}(\mathbf{v})$, in which the point $\mathbf{e}'$ is either the current viewpoint $\mathbf{e}$ or the anticipated future viewpoint $\mathbf{e} + gtime\ \Delta\mathbf{e}$ depending on whether $\Delta\mathbf{e}\cdot\vec{\mathbf{e}}$ is negative or positive respectively (i.e. viewer moving backwards or forwards).

Discussion. Selectively refined meshes within the VDPM framework, and in particular those resulting from geomorphs, may have thin (near-degenerate) triangles. Several papers warn against using such meshes for LOD rendering. In our opinion this is unjustified. Thin triangles can produce unsightly artifacts when they result from inaccurate computation of approximation error; however, such artifacts are avoided when computing exact error as in Section 5.1. Some authors have expressed concern that thin triangles may misbehave during rendering. However, we have not observed any rasterization artifacts on the graphics platforms we have tested (SGI Maximum Impact and Infinite Reality), as demonstrated on the accompanying video.

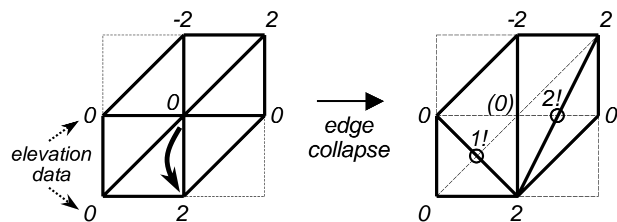


Figure 7: For this edge collapse, evaluating the maximum height deviation solely at grid points gives an error of zero.

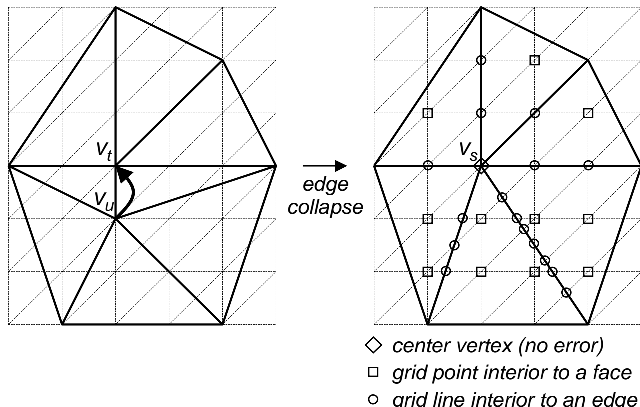


Figure 8: For correct approximation error, it is crucial to consider all vertices in the union partition of the two triangulations.

5 SPECIALIZATION OF VDPM TO TERRAINS

In this section we discuss how the VDPM framework can be specialized to address the rendering of height fields.

Experiments reveal that backface simplification presents little benefit. For instance, in Figure 11 it does not result in any coarsening of the mesh. We therefore omit it as a view-dependent refinement criterion by simplifying the function is_invisible as follows:

```
function is_invisible(vs)
  return outside_view_frustum(vs)
```

Since the texture image is mapped onto the terrain using a vertical projection, we measure surface deviation *parametrically* using strictly a vertical distance δ_v as done in [18]. If in addition, texture mapping is performed without Gouraud shading, the storage of vertex normals becomes unnecessary.

As a consequence, we omit storing the fields $\hat{\mathbf{n}}_v$, α_v, and μ_v (as highlighted by the '†' symbol in Figure 4), thereby reducing static storage to $48n$ bytes using 16-bit indices, quantization, and lookup tables as described in Section 4.1.

5.1 Exact approximation error

For height fields, it is common in the literature to measure the approximation error of a simplified mesh by its maximum vertical deviation at the original grid points (e.g. [2, 7, 11]). For view-dependent LOD, however, measuring deviation solely at grid points is generally insufficient, which is surprising at first since the only input is the discrete set of grid points, i.e., there is no knowledge of the surface between the points.

For the edge collapse transformation illustrated in Figure 7, both meshes (before and after the transformation) interpolate the grid points, yet they have different geometric shapes. Even if it is argued that both interpolants are equally valid, clearly the two rendered meshes look different, and the mesh transformation may lead to

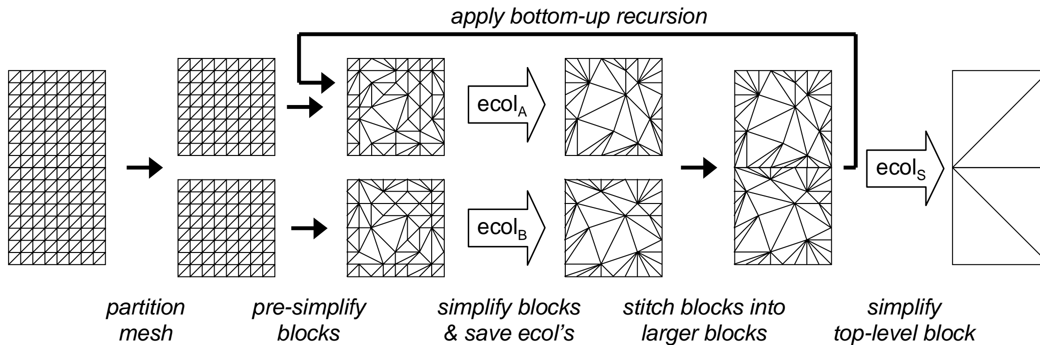


Figure 9: Steps in the hierarchical block-based simplification done as a preprocess.

an arbitrarily large pop. This point is more than academic; an initial implementation using the naive approximation error did in fact result in unexpectedly large pops, as demonstrated qualitatively on the video. Even though geomorphs hide such pops, it is still useful to have an accurate estimate of screen-space error.

The solution is to measure *maximum* (L^∞) approximation error with respect to a reference surface.[1] A natural choice for this reference surface is the regular triangulation of the grid points. We therefore want to compute the maximum height deviation between this triangulated grid and the open neighborhood of each edge collapse transformation. (Similar derivations are described in [1, 3, 9].) The maximum height deviation between two triangle meshes is known to lie at a vertex of their *union partition* in the plane (e.g. the vertices labeled in Figure 8). An efficient way to enumerate the vertices of the union partition is to consider: (1) the grid points internal to the faces adjacent to v_s, and (2) the grid line crossings internal to the edges adjacent to v_s. Note that the computed error is not just an upper-bound, it is exact, and it is always computed with respect to the original fully detailed mesh. This error, computed during the preprocessing discussed in the next section, is stored in the Vsplit field δ_v for use in the runtime criterion of Section 4.2.

It should be pointed out that for regular subdivision schemes based on quadtrees and bintrees, all grid line crossings happen to fall exactly on grid points, so the naive approach is in fact sufficient.

5.2 Hierarchical PM construction

We develop a hierarchical scheme for constructing PM representations of large terrains. The scheme, applied as a preprocess, partitions surface geometry into blocks and uses bottom-up recursion to simplify and merge the block geometries. Our approach is motivated by three considerations:

- Because simplification methods start from a detailed mesh and successively remove vertices, they are inherently memory-intensive. Although the mesh of Figure 1 does have an acceptable approximation that fits in main memory, attempting to form this approximation by simplifying the mesh as a whole would be impractical. For height fields, an alternative approach requiring less memory is to start from a coarse approximation and progressively insert vertices [2, 11]. However, such greedy refinement methods generally yield inferior approximations (e.g. compare Figures 25-26 in [11] with Figure 8b in [15]). Our hierarchical strategy allows us to tackle the problem piecemeal with an accurate simplification-based method.
- For even larger models, a pre-simplified mesh may still be too large to fit in main memory. If one resorts to the operating system's virtual memory manager, the resulting paging causes the process to pause intermittently, disturbing frame rate. By partitioning the refinement database into a block hierarchy, we can exploit domain knowledge to explicitly pre-fetch refinement data before it is needed.
- Just as the geometry data may be too large for memory, so may its associated texture image. Clip-maps [20] offer an elegant solution to this problem, but require hardware assistance currently available only on high-end systems. A more traditional approach is to partition the texture image into *tiles* that can be mip-mapped and paged independently. In our scheme, a texture tile may be associated with each block, as the PM construction can optionally guarantee that mesh faces never cross block boundaries.

[1] The use of an L^2 norm to integrate squared error over the domain is inadequate, as reported in [11].

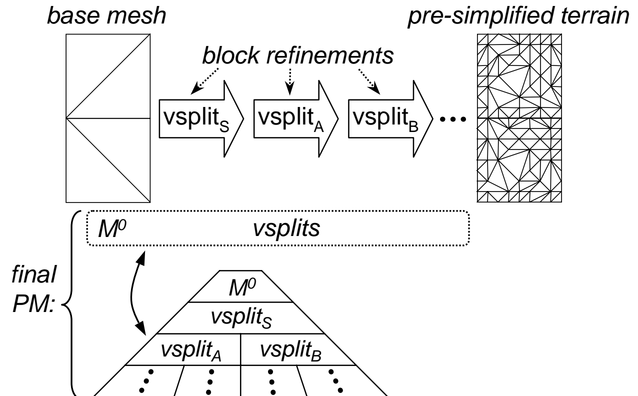
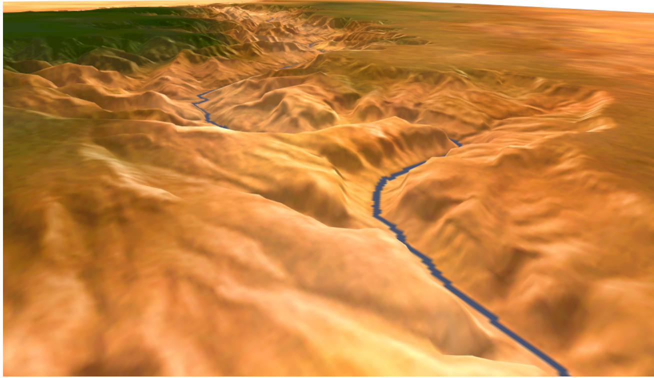


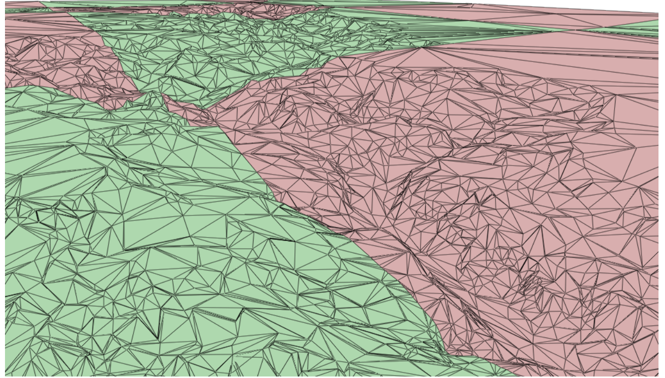
Figure 10: Result of the hierarchical construction process.

After partitioning the model into blocks, the recursive scheme proceeds as illustrated in Figure 9. Starting at the lowest level, it simplifies each block by iteratively applying a sequence of *ecol* transformations. This sequence is chosen by selecting at each iteration the *ecol* giving rise to the lowest approximation error, using the metric of Section 5.1. The simplification process terminates when the approximation error of the next best *ecol* exceeds a user-specified threshold for the current level. In order to avoid refinement dependencies between adjacent blocks, we constrain *ecol*'s to leave boundary vertices untouched. Above the first level, if texture tiles are desired, we also constrain *ecol*'s to prevent displacement of tile boundaries within the block, as shown by the checkerboard pattern in Color Plate 1d-e. The simplification sequences (e.g. $ecol_A$, $ecol_B$) are saved to disk. Then, the resulting simplified meshes are stitched together 2×2 at a time, and the process is repeated at the next higher level using these larger blocks.

Special treatment is given to the first and last levels of simplification. In the first level, we discard for each block all initial *ecol*'s



(a) Texture mapped



(b) Underlying triangle mesh

Figure 11: One frame from the 30 frame/sec flyover of the terrain shown in Figure 1. The screen-space error tolerance τ is 2.1 pixels for a 720x510 window. The active mesh has 12,154 faces and 6,096 vertices. The fraction of vertices undergoing geomorph refinement and coarsening is 26.8% and 0.2% respectively. Note how the sizes and shapes of the triangles adapt to the complex topography.

Level	max. error (range 0–255)	number of faces ($2n$)
original mesh	0.0	16,777,216
pre-simplified	0.5	1,453,154
level 0	1.0	540,604
level 1	2.0	215,064
level 2	∞ (106.0)	64

Table 1: PM hierarchy statistics.

until a user-specified error tolerance is exceeded. Effectively this amounts to truncating the final hierarchy but it avoids the intermediate storage costs. In the last level, in which there is only a single block, we permit simplification of the mesh boundary, since inter-block dependencies are no longer a concern. Thus the final base mesh M^0 consists of only 2 triangles, or 2 triangles per tile if texture tiles are desired (Color Plate 1e).

To form a hierarchical PM, we invert each recorded *ecol* sequence to form a *vsplit* sequence called a *block refinement* (e.g. $vsplit_S$, $vsplit_A$, and $vsplit_B$ in Figure 10). We concatenate these block refinements and store them along with the base mesh. As in the construction of an ordinary PM, this final assembly involves renumbering the vertex and face parameters in all *vsplit* records. However, the renumbering only requires depth-first access to the block refinements within the hierarchy, so that memory usage is moderate.

Although the hierarchical construction constrains simplification along block boundaries at lower levels, inter-block simplification occurs at higher levels, so that these constraints do not pose a significant runtime penalty. For our flyover example in Section 6, and for a given pixel tolerance, a non-hierarchical PM representation reduces the average number of active faces by only 0.8%.

As mentioned above, the user specifies a threshold for the maximum approximation error at each simplification level. We choose these thresholds so that block refinements at all levels have roughly the same number of *vsplit* refinements. For example, in Color Plate 1 the thresholds are 0.03%, 0.04%, 0.1%, and ∞, expressed as fractions of the terrain width. Since these thresholds form an upper-bound on the errors of all *vsplit*'s below that level, they can be used to determine which block refinements need to be memory-resident based on the current viewing parameters, and which others should be prefetched based on the anticipated view changes. Specifically, we can use the thresholds to compute the maximum screen-projected error for each active block as in [18]. If this error exceeds the screen-space tolerance τ, its child block refinements are loaded into memory and further tested.

Since block refinements correspond to contiguous sequences in the *vertices*, *faces*, and *vsplits* arrays of Figure 4, we can reserve virtual memory for the whole PM (i.e. the entire arrays), and use sparse memory allocation to commit only a fraction to physical memory. For instance, the Microsoft Windows operating system supports such a reserve/commit/decommit protocol using the VirtualAlloc() and VirtualFree() system calls.

6 RESULTS

Preprocessing. The 4,097 $\times$ 2,049 grid of Figure 1 represents actual elevation data and satellite imagery near the Grand Canyon. The elevation at each grid vertex is given as an integer in the range 0–255, where one unit represents 10 meters. To make the example more challenging, we exaggerated the elevation by 40%, assigning 14 meters to each unit. We partitioned the initial mesh into 8 $\times$ 4 blocks of 513 $\times$ 513 vertices, and applied the hierarchical construction scheme of Section 5.2. Table 1 shows the maximum error tolerance for each level of the hierarchy, in the original 0–255 scale, together with the total number of faces at that level. As the meshes are too dense to visualize, Color Plate 1 shows a much simpler example with 4 $\times$ 4 blocks of 33 $\times$ 33 vertices.

We pre-simplify the blocks up to an error of half the original data resolution (0.5 in the 0–255 range). With a resulting pre-simplified mesh of n = 732,722 vertices, the static data structures use 49.8 MB. (On low-end platforms where storage is at a premium, the pre-simplification error threshold can be increased to 1.0 to further reduce size to 13.1 MB.) In contrast, the dynamic data structures require only 0.6 MB of memory in a typical situation such as that in Figure 11 (where the number of active vertices m = 6,000 and the number of morphing vertices g = 1,500).

Runtime. In the accompanying video, we demonstrate a 2-minute flyover of the Grand Canyon terrain. In world units, speed during the flight is approximately Mach 10. The flyover is rendered in real-time on an SGI Octane workstation (single processor 195 MHz R10K with Maximum Impact graphics), in a window of 710x520 pixels. We obtain a constant frame rate of 30 frames/sec by regulating the screen-space error tolerance τ to maintain approximately 12,000 active faces, and by amortizing the work of the procedure adapt_refinement over 3 frames, as described in [16]. Figure 11 shows one frame from the flyover, in which τ equals 2.1 pixels. Over the whole flight, the screen-space tolerance τ averages 1.7 pixel, and attains a maximum of 3.3 pixels. (Because of approximations in the computation of screen-space errors [16], the tolerance values τ are unfortunately not upper bounds on screen-space error.) The fraction of vertices undergoing geomorph refinement and coarsening averages 28.3% and 2.2% respectively.

Scheme	hardware config.	window dimensions	frames/ sec	# faces ($2m$)	pixel tol. τ avg.	pixel tol. τ max.
ours	R10K-MXI	710x520	30	12,000	1.7	3.3
”	R10K-MXI	710x520	60	5,000	3.5	8.3
”	R10K-MXI	710x520	72	4,000	4.3	10.0
”	Onyx-IR	710x520	30	16,000	1.3	2.3
”	Onyx-IR	710x520	60	8,000	2.1	4.5
”	R10K-MXI	1000x1000	30	11,000	2.9	5.2
[8]	R10K-MXI	1000x1000	30	3,000	n/a	$\approx$6.0
”	Onyx-IR	1000x1000	30	6,000	n/a	n/a
[18]	Onyx-RE2	640x480	20-30	4-9,000	$\approx$2.0	$\approx$2.0

Table 2: Runtime statistics and comparison with previous work.

By modifying the regulation to instead maintain 5,000 active faces, we consistently achieve a target frame rate of 60 frames/sec. Similarly, 4,000 active faces yields a rate of 72 frames/sec. Table 2 lists statistics for these flyovers, as well as similar ones recorded on an SGI Onyx Infinite Reality system. As we currently only make use of a single CPU, performance gains on the Onyx are limited.

For comparison, Table 2 also lists statistics for some previous methods. Unfortunately, the terrains are different in each case, thus preventing direct comparisons. The quality of our results are due to the combination of three advantages:

1. The framework is efficient; by performing output-sensitive, incremental work, it is able to adapt and render an active mesh of 12,000 faces at 30 frames/second on a uni-processor workstation.
2. These mesh faces are derived from geometrically optimized refinement transformations, thus providing an accurate screen-space approximation.
3. Through temporal coherence, geomorphs mask the remaining screen-space approximation error.

7 GENERALIZATION: ARBITRARY MESHES

Both the output-sensitive data structures (Section 4.1) and the runtime geomorph framework (Section 4.2) apply to VDPM representations of arbitrary meshes, as demonstrated on the accompanying video. In this section we briefly discuss how the techniques of Section 5 can be generalized to work on arbitrary meshes.

Approximation error. Several recent methods are able to track upper bounds on maximum geometric error during simplification of arbitrary meshes. Among these, at least two [3, 13] specifically consider sequences of edge collapse transformations. It would be easy to include these methods within the general VDPM framework. Other recent work [4] obtains bounds on parametric approximation error, which is appropriate in the presence of texture mapping.

Hierarchical construction. A Voronoi construction as in [9] can be used to recursively partition an arbitrary mesh into a hierarchy of regions. Although the regions are no longer square blocks, the bottom-up simplification and stitching scheme should work with little modification.

ACKNOWLEDGMENTS

I wish to thank Anuj Gosalia of the Microsoft Direct3D group for helpful feedback, and Cindy Grimm, John Snyder, and Rick Szeliski for comments on the paper.

REFERENCES

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

[2] CIGNONI, P., PUPPO, E., AND SCOPIGNO, R. Representation and visualization of terrain surfaces at variable resolution. *The Visual Computer 13* (1997), 199–217.

[3] COHEN, J., MANOCHA, D., AND OLANO, M. Simplifying polygonal models using successive mappings. In *Visualization '97 Proceedings* (1997), IEEE, pp. 81–88.

[4] COHEN, J., OLANO, M., AND MANOCHA, D. Appearance-preserving simplification. *Computer Graphics (SIGGRAPH '98 Proceedings)* (1998).

[5] COHEN-OR, D., AND LEVANONI, Y. Temporal continuity of levels of detail in Delaunay trianulated terrain. In *Visualization '96 Proceedings* (1996), IEEE, pp. 37–42.

[6] DE FLORIANI, L., MAGILLO, P., AND PUPPO, E. Building and traversing a surface at variable resolution. In *Visualization '97 Proceedings* (1997), IEEE, pp. 103–110.

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

[8] DUCHAINEAU, M., WOLINSKY, M., SIGETI, D., MILLER, M., ALDRICH, C., AND MINEEV-WEINSTEIN, M. ROAMing terrain: real-time optimally adapting meshes. In *Visualization '97 Proceedings* (1997), IEEE, pp. 81–88.

[9] ECK, M., DEROSE, T., DUCHAMP, T., HOPPE, H., LOUNSBERY, M., AND STUETZLE, W. Multiresolution analysis of arbitrary meshes. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 173–182.

[10] 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.

[11] GARLAND, M., AND HECKBERT, P. Fast polygonal approximation of terrains and height fields. CMU-CS 95-181, CS Dept., Carnegie Mellon University, 1995.

[12] GARLAND, M., AND HECKBERT, P. Surface simplification using quadric error metrics. *Computer Graphics (SIGGRAPH '97 Proceedings)* (1997), 209–216.

[13] 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.

[14] HECKBERT, P., AND GARLAND, M. Survey of polygonal surface simplification algorithms. In *Multiresolution surface modeling (SIGGRAPH '97 Course notes #25)*. ACM SIGGRAPH, 1997.

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

[16] HOPPE, H. View-dependent refinement of progressive meshes. *Computer Graphics (SIGGRAPH '97 Proceedings)* (1997), 189–198.

[17] LILLESKOG, T. Continuous level of detail. Master's thesis, Department of Computer Science, Norwegian University of Science and Technology, February 1998.

[18] 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.

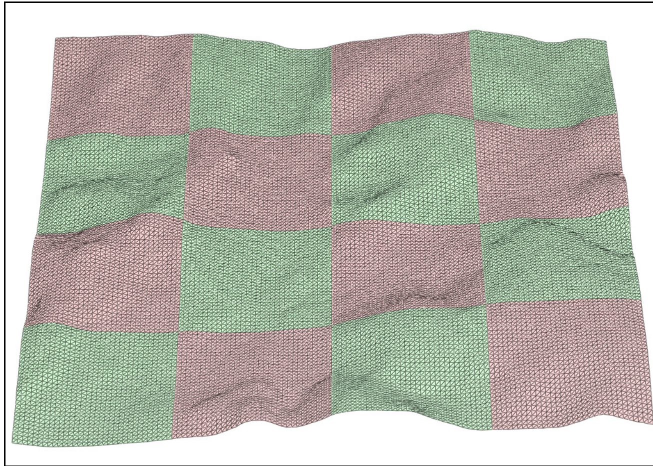
[19] LUEBKE, D., AND ERIKSON, C. View-dependent simplification of arbitrary polygonal environments. *Computer Graphics (SIGGRAPH '97 Proceedings)* (1997), 199–208.

[20] MONTRYM, J., BAUM, D., DIGNAM, D., AND MIGDAL, C. InfiniteReality: a real-time graphics system. *Computer Graphics (SIGGRAPH '97 Proceedings)* (1997), 293–302.

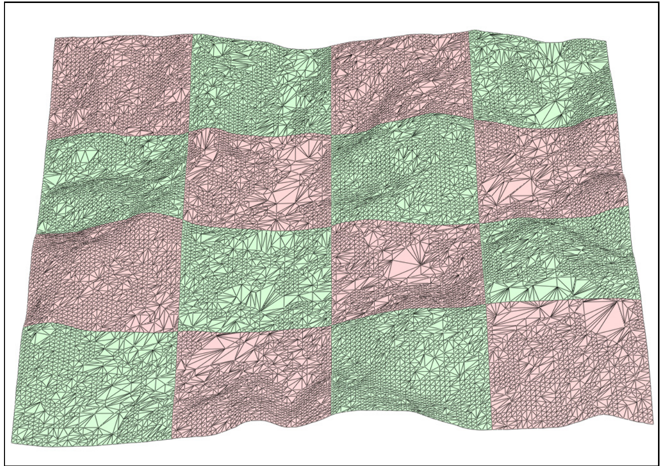
[21] ROSSIGNAC, J., AND BORREL, P. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics*, B. Falcidieno and T. L. Kunii, Eds. Springer-Verlag, 1993, pp. 455–465.

[22] 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.

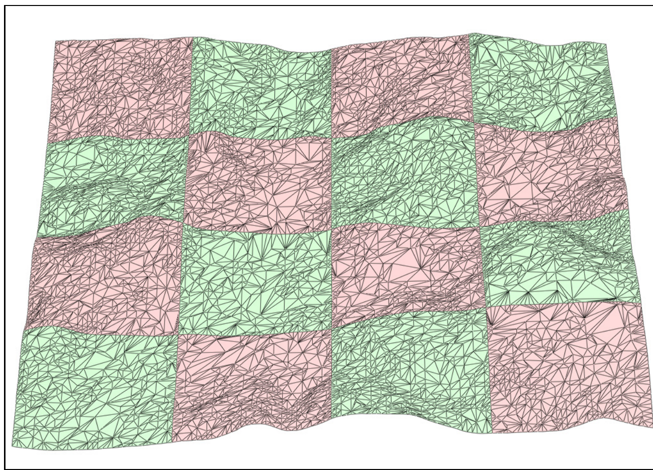
[23] XIA, J., AND VARSHNEY, A. Dynamic view-dependent simplification for polygonal models. In *Visualization '96 Proceedings* (1996), IEEE, pp. 327–334.



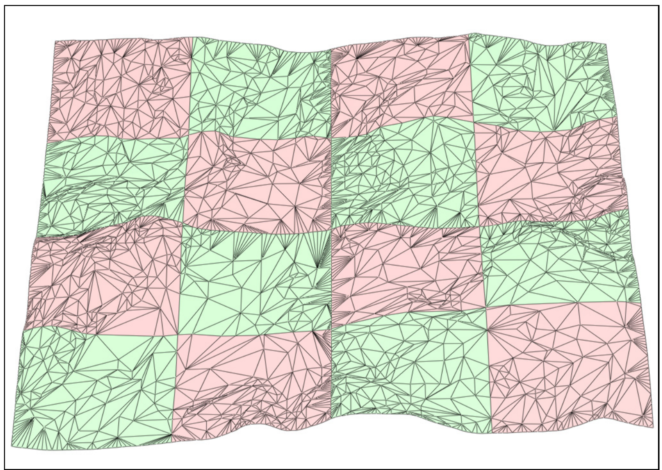
(a) Original mesh (129 × 129 vertices; 32,768 faces; max δ=0)



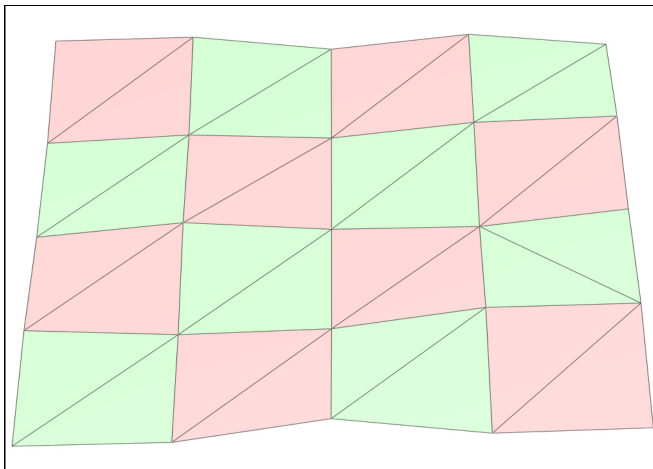
(b) Pre-simplified mesh (21,622 faces; max δ=0.03%)



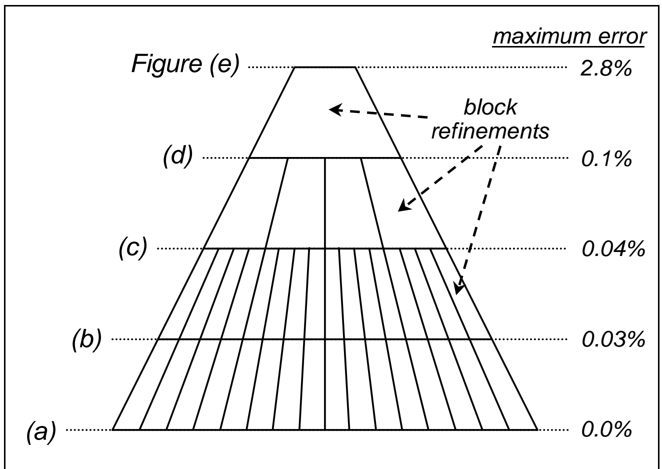
(c) Level 0 simplification (11,048 faces; max δ=0.04%)



(d) Level 1 simplification (3,594 faces; max δ=0.1%)



(e) Level 2 — base mesh M^0 (32 faces; max δ=2.8%)



(f) PM hierarchy showing block refinements

Color Plate 1: The hierarchical PM construction process partitions the model into blocks and recursively simplifies and combines the blocks. The result is a base mesh M^0 and a hierarchy of block refinements. The value max δ indicates the maximum deviation of the meshes at the given level, as a fraction of the terrain width.