Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization

Peter Lindstrom, *Member*, *IEEE*, and Valerio Pascucci, *Member*, *IEEE*

Abstract—This paper describes a general framework for out-of-core rendering and management of massive terrain surfaces. The two key components of this framework are: view-dependent refinement of the terrain mesh and a simple scheme for organizing the terrain data to improve coherence and reduce the number of paging events from external storage to main memory. Similar to several previously proposed methods for view-dependent refinement, we recursively subdivide a triangle mesh defined over regularly gridded data using *longest-edge bisection*. As part of this single, per-frame refinement pass, we perform triangle stripping, view frustum culling, and smooth blending of geometry using geomorphing. Meanwhile, our refinement framework supports a large class of error metrics, is highly competitive in terms of rendering performance, and is surprisingly simple to implement. Independent of our refinement algorithm, we also describe several data layout techniques for providing coherent access to the terrain data. By reordering the data in a manner that is more consistent with our recursive access pattern, we show that visualization of gigabyte-size data sets can be realized even on low-end, commodity PCs without the need for complicated and explicit data paging techniques. Rather, by virtue of dramatic improvements in multilevel cache coherence, we rely on the built-in paging mechanisms of the operating system to perform this task. The end result is a straightforward, simple-to-implement, pointerless indexing scheme that dramatically improves the data locality and paging performance over conventional matrix-based layouts.

Index Terms—Terrain visualization, surface simplification, view-dependent refinement, continuous levels of detail, edge bisection, error metrics, geomorphing, out-of-core algorithms, external memory paging, data layouts.

1 Introduction

View-dependent refinement and out-of-core data management are two critical components of large-scale, interactive visualization of massive terrain surfaces. In recent years, several effective yet quite complicated, often specialized, and many times incompatible methods have been proposed for these two tasks. Whereas large-scale terrain visualization was once synonymous with industrial flight simulation, a plethora of emerging uses, ranging anywhere from military and scientific applications to video games and hobby use, suggest that simple-to-implement yet powerful algorithms for terrain visualization are becoming increasingly valuable. In part to address this problem, we recently proposed a general framework for performing highly interactive view-dependent rendering, as well as a transparent mechanism for improving multilevel cache performance and enabling efficient paging of gigabyte-size data sets [1]. In this paper, we provide an extended overview and in-depth discussion of these algorithms.

We will first describe an algorithm for efficient view-dependent refinement. Using the common vertex hierarchy induced by recursive edge/triangle bisection [2], [3], [4], we show that it is possible to:

1. construct an adaptive mesh from scratch each frame,
2. perform fast, hierarchical view frustum culling,
3. create smooth transitions in the geometry using geomorphing, while
4. simultaneously outputting a single generalized triangle strip for the entire mesh that can be efficiently rendered.

Moreover, all of these tasks can be performed without having to maintain any state information, except, of course, for the output being generated. That is, the traversal can be cast in a purely functional form, which not only makes efficient implementations possible, but is also a feature that meshes well with the out-of-core component of our framework. Because no state information is associated with the mesh vertex data, we can access this data in a read-only fashion. This improves CPU cache performance and also allows the on-disk data to be efficiently *memory mapped* without the need for frequent write-back of dirty pages.

As already alluded to, the external memory component of our system is based on associating the on-disk terrain database with a large region of read-only logical address space, which may greatly exceed the amount of physical memory. Using this approach, on-demand data paging is performed transparently by the operating system. Instead of focusing on explicit paging mechanisms, we leave this as an open issue and instead discuss different schemes for rearranging the terrain data so that it can be accessed in a cache coherent manner. We will describe the problem of coherent data layouts in the second part of our paper.

Because our method is stateless, we do not need to maintain dependencies in the vertex hierarchy [2] nor do

• *The authors are with the Lawrence Livermore National Laboratory, 7000 East Ave., L-560, Livermore, CA 94551. E-mail: {pl, pascucci}@llnl.gov.*

Manuscript received 15 Feb. 2002; revised 15 Mar. 2002; accepted 2 Apr. 2002.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 116205.

we make explicit use of frame-to-frame coherence using mechanisms like priority queues [3], [5], active cuts [2], [3], [5], [6], or multiframe amortized evaluation [6]. We do not suggest that such techniques are not useful; however, making successful use of these concepts considerably complicates implementations and we have seen no evidence that our top-down approach cannot perform as well as, or even better than, more complicated previously published methods.

Another feature of our framework is that its individual components are modular—it is, for example, possible to add, remove, or even swap out components such as triangle stripping, culling, geomorphing, data indexing, etc., without having to perform significant code surgery. In addition, adding any one of these components does not change the required on-disk data structures. Rather, the per-vertex terrain data is limited to position (or just elevation), a scalar error term, and the radius of a bounding sphere. We anticipate that this modularity will aid in quickly implementing the core feature set of our refinement algorithm.

Many of the details of our framework were presented in [1]. We here provide a more thorough exposition, but also significant new material. The major contributions of this paper over [1] include:

1. An easy-to-integrate technique for position-based geomorphing. The morphs are driven by the screen space error for a vertex, which ensures that the terrain geometry is determined entirely by the camera view and can be varied smoothly with the viewpoint.
2. An extended discussion and derivation of alternative error metrics.
3. Derivations of all index computations needed for hierarchical traversal.
4. Additional experimental results.

We include performance data and animations showing the quality of our geomorphs and analyze and compare different error metrics. Finally, we have attempted to further clarify the steps in our algorithms to facilitate their implementation and to make the transfer between abstract concepts and actual code as straightforward as possible.

2 Previous Work

In this section, we discuss related work in large-scale terrain visualization. We will focus particularly on algorithms for view-dependent refinement of terrain and schemes for out-of-core paging and memory coherent layout of multiresolution data.

2.1 View-Dependent Refinement

Over the last several decades, there has been extensive work in the area of terrain visualization and level of detail (LOD) creation and management. We will here limit our discussion to the more recent work on view-dependent simplification and refinement of terrain surfaces.

Gross et al. [7] were among the first to propose a method for adaptive mesh tessellation at near interactive rates. Their technique is based on a wavelet transform of the gridded data from which large detail coefficients are chosen for selective refinement. A windowing technique is also described that allows some regions of the mesh to be more refined than others. Lindstrom et al. [2] describe an algorithm for interactive, view-dependent refinement of terrain. They represent the terrain as a mesh with subdivision connectivity that is locally refined using recursive *edge bisection*. The algorithm conceptually works bottom-up, by recursively merging triangles until a screen space error tolerance is exceeded. In actuality, the terrain is partitioned into a quadtree of rectangular blocks of vertices. Taking advantage of frame-to-frame coherence, the active cut in this quadtree is visited and updated, after which individual vertices within each block are considered for insertion or removal. Due to this blocking of the terrain, special care must be taken to ensure that no cracks form between the blocks. Handling this problem in the context of asynchronous paging of blocks is nontrivial and enforcing dependencies between vertices can be costly.

Hoppe extended his work on *progressive meshes* to allow view-dependent refinement of arbitrary meshes [6]. This technique was later specialized for terrain rendering [8]. The runtime performance reported by Hoppe places his method among the fastest ones published to date. However, the memory requirements of his method, while lower than in [6], are still considerable. In addition, fully implementing his algorithm is not an easy task.

Using the same space of meshes as in [2], Duchaineau et al. [3] proposed several improvements over Lindstrom et al.'s method in their ROAM algorithm. Instead of organizing the mesh as an acyclic graph of its vertices, they suggest using a binary tree over the set of triangles. Using this data structure, crack prevention is made easier. Another significant contribution is the idea of maintaining two queues for split and merge operations, which allows incremental changes to the mesh to be made in order of importance, while also allowing the refinement to be preempted whenever a given time budget is reached. Unfortunately, robustly implementing the dual-queue algorithm, not to mention the many other components of their method, has proven difficult.

Several other algorithms based on edge bisection have since been published, with different strengths and weaknesses in terms of visual accuracy and memory and time complexity [4], [5], [9], [10], [11]. Gerstner [11] and Pajarola [4] both discuss how to remove some of the dependencies in the vertex hierarchy by implicitly coding them into the object space errors, but do not extend this concept to view-dependent metrics. Similar to [2], efficient rendering is achieved in [4] by organizing the set of triangles into a single generalized triangle strip that follows the Sierpinski space filling curve. We, too, use a single triangle strip in our refinement algorithm. Röttger et al. [9] present a memory-efficient solution to terrain rendering, requiring only two bytes of storage per vertex, but their approach relies heavily on a particular view-dependent metric that approximates Euclidean distances with the Manhattan distance. We will revisit some of these methods briefly in the sections below and contrast them with our own method.

2.2 Geomorphing

The view-dependent level-of-detail algorithms discussed so far have the ability to adapt the terrain mesh at the granularity of individual vertices. Even though this allows making fine-scale changes to the mesh from one frame to the next, these changes, if geometrically large enough, can lead to temporal artifacts known as "popping." *Geomorphing*, or just *morphing*, is a common approach to counter such visually disturbing phenomena by interpolating the geometric transitions between different levels of detail smoothly over time [12].

Cohen-Or and Levanoni [13] proposed using transition zones, based on the distance to each vertex, to blend the geometry of Delaunay-triangulated terrain. Such a distance-based approach was also advocated by Pajarola [4]. Willis and coworkers described a similar technique that was used in the popular IRIS Performer visual simulation toolkit [14]. Hoppe took a different approach by explicitly animating vertex splits and edge collapses over time. Because of the inherent dependencies between vertices in the hierarchy, his time-based geomorphs imposed somewhat complicated restrictions on when a vertex could be removed. Duchaineau et al. [3] suggested using a similar time-based morphing strategy for their ROAM algorithm.

A slightly simpler and, in a sense, more disciplined approach than purely time or distance-based morphing is to make direct use of the given error metric to parameterize the morphs. In this way, the screen space error is used as the parameter that feeds into the interpolation. In the algorithm by Röttger et al. [9], the normalized error term that was used to make refinement decisions was also used as a parameter for morphing the geometry. More recently, Cline and Egbert [15] proposed using a similar approach to morph a quadtree representation of the terrain. For each quadtree patch, they determine a continuous, view-dependent level-of-detail parameter and use its fractional part to interpolate between the two closest, discrete level-of-detail representations. Our approach to geomorphing is similar in spirit to [9], [15], but we use the actual screen space error as the morph parameter and blend the geometry when this error falls within a user-specified range.

2.3 Out-of-Core Paging and Data Layout

External memory algorithms [16], also known as out-of-core algorithms, address issues related to the hierarchical nature of the memory structure of modern computers (fast cache, main memory, hard disk, etc.). Managing and making the best use of the memory structure is important when dealing with large data sets that do not fit in main memory. In most terrain visualization systems [2], [3], [4], [8], [17], [18], [19], [20], the external memory component is essential for handling real terrain and GIS databases. Hoppe [8] addresses the problem of constructing a progressive mesh of a large terrain using a bottom-up scheme by decomposing the terrain into square tiles that are merged after independent decimation and which are then further simplified. Döllner et al. [21] address the issue of external memory handling of large textures for terrain visualization. Reddy et al. [19] implemented a custom VRML browser specialized for terrain visualization, where efficiency is gained by combined use of multiresolution tiling, data caching, and predictive prefetching. The out-of-core component of the terrain system presented by Pajarola [4] is based on a decomposition of the domain into square tiles, which are stored in a database that supports fast 2D range queries.

Whereas the prevailing strategy for terrain paging has been to split the terrain up into rectangular tiles of varying resolution that are paged in on demand and to optimize the size of these tiles and the I/O path from disk to memory, our approach is instead to optimize the data layout to improve the memory coherency—both in-core and out-of-core—for a given access pattern. This approach is, in a sense, orthogonal to the manner in which the data is paged in. For simplicity, we leave it to the operating system to perform this task.

Balmelli [22] uses the Z-order (also called Morton order) space filling curve to efficiently navigate a pointerless quadtree data structure. He uses simple expressions for computing neighbor and nearest common ancestor relations between nodes, allowing fast generation of edge bisection triangulations. The use of the Z-order curve for traversal of quadtrees [23] has also proven useful for speeding up matrix operations [24].

Recently, Pascucci [25] introduced a simple address transformation that turns a single-resolution indexing scheme into a multiresolution version, which is optimized for coarse-to-fine breadth-first traversal. This technique has proven effective for visualizing large 3D rectilinear grids [26]. The downside of this scheme is the need to apply the address transformation for each data access. The data layout schemes developed in this paper are inspired by this technique, but have more stringent performance requirements. In particular, we show how to speed the address computation up based on context. Another hierarchical address computation for gridded data was introduced by Gerstner [11]. In this work, the bintree hierarchy of triangles induced by the Sierpinski space filling curve is used to compute the vertex indices during runtime. The locality of the index is inherited from the Sierpinski curve. The application of this address to our case does not seem appropriate because of the scattering of unused space that results from duplicating vertex addresses.

In this paper, we present three different pointerless hierarchical data layouts that have been shown to improve the cache and paging efficiency by orders of magnitude over more naive layouts. We draw upon previous work on quadtree and space filling curve layouts, but leverage the fact that the data access pattern is given by a top-down recursive traversal of the height field vertices. We will begin by explaining how this recursive traversal is used for constructing adaptive meshes at runtime.

3 View-Dependent Refinement

The goal of view-dependent level-of-detail algorithms is to construct a mesh with a small number of triangles that, for a given view, is a good approximation of the original, full-detail mesh. This construction is done continuously at runtime and, whenever the viewpoint changes, the mesh is updated to reflect the change. To measure how well the coarse mesh approximates the original, it is common to measure the *object space* error ϵ between the original mesh and its approximation, e.g., as the vertical deviation between corresponding points, and to project this error

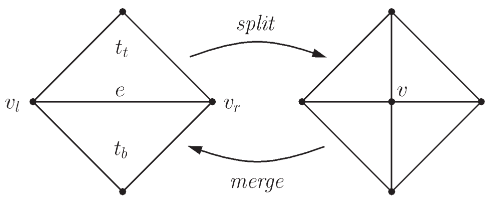


Fig. 1. Split edge $e = \{v_l, v_r\}$ and diamond $T = \{t_b, t_t\}$ of a vertex v.

onto the screen, e.g., using perspective projection, to obtain a view-dependent measure of error $\rho(\epsilon)$. Depending on whether the mesh is *simplified* bottom-up (fine-to-coarse) or *refined* top-down (coarse-to-fine), triangles are merged or split to ensure that the projected errors meet some tolerance or the mesh meets a given triangle budget.

The input to our refinement algorithm is a uniformly sampled height field, i.e., a rectilinear grid of elevations. Formally, the height field can be represented as a function $z(x, y)$ over a 2D domain. To form a continuous surface, we use linear interpolation between samples, resulting in a piecewise linear triangle mesh. By selecting only a subset of points from the height field, a coarser mesh is obtained. It is this selection in particular that we will be concerned with below.

Below, we present a framework for performing top-down, view-dependent refinement of the terrain mesh. We show how a single procedure can be used to efficiently refine the mesh, blend transitions in the geometry using geomorphing, cull the mesh against the view volume, and simultaneously build a single (generalized) triangle strip for the entire mesh. This procedure makes no use of frame-to-frame coherence, but rather builds the mesh from scratch for each individual frame. We first describe our main approach to refinement and follow with details of how to implement each of its components.

3.1 Longest Edge Bisection

There are two important classes of meshes used for view-dependent refinement: unstructured meshes (sometimes called triangulated irregular networks or TINs) [8], [13], [28], [27], [29], and regular (or semiregular) meshes with subdivision connectivity [1], [2], [3], [4], [5], [9], [11]. Whereas TINs have the potential to represent a surface with fewer triangles for a given error tolerance (see, for example, [5] for a quantitative analysis), the simplicity of regular subdivision hierarchies makes them more appropriate for our purpose.

We make use of a particular type of subdivision based on *longest edge bisection* [2], [3], [9], [11]. The meshes produced by this subdivision scheme, also called 4-*k* meshes [30], right-triangulated irregular networks [5], and restricted quadtree triangulations [4], have the property that they can be refined locally to selectively resolve features (see Fig. 3, for example). In the edge bisection scheme, an isosceles right triangle is refined by bisecting its hypotenuse, thus creating two smaller right triangles (Fig. 1). For the vertex v inserted in this refinement step, we call the bisected edge the *split edge* e_v of v. The two triangles (or single triangle in the case of split edges on the boundary) that share e_v are called the *diamond* T_v of v [3]. The split edge and diamond are illustrated in Fig. 1. Starting with a coarse base mesh (typically two or four triangles), an adaptive, recursive refinement of the mesh is made (Fig. 2). The refinement criterion, i.e., whether to insert a vertex, is generally based on whether the vertex's diamond approximates the corresponding part of the full-resolution mesh well enough. For *view-dependent* refinement, this criterion also depends on factors such as the position of the viewer relative to the vertex.

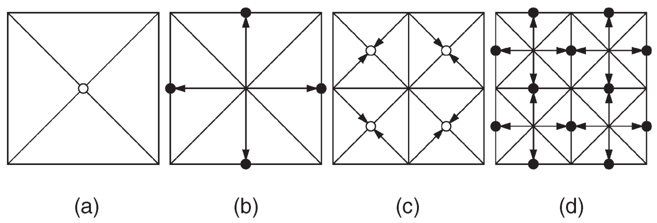


Fig. 2. Edge bisection hierarchy. The arrows correspond to parent-child relationships in the directed acyclic graph of mesh vertices.

As is evident from Fig. 2, the vertices introduced in the subdivision map directly to points on a rectilinear grid. Thus, it is natural to use the edge bisection hierarchy as a multiresolution representation for approximating height fields. As in other methods based on edge bisection, the dimensions of the underlying grid are constrained to $2^{n/2} + 1$ vertices in each direction, where n is the (even) number of refinement levels.

The mesh produced by edge bisection can be represented as a *directed acyclic graph* (DAG) of its vertices. A directed edge (i, j) from i to one of its DAG children j corresponds to a triangle bisection in which j is inserted on the hypotenuse and connected to the apex at i (Fig. 2). Thus, all nonleaf vertices not on the boundary of the mesh have four children and two parents in the DAG. Boundary vertices have two children and one parent. For a given refinement M of a mesh, we say that a vertex is *active* if it is included in M. Furthermore, M is *valid* if it forms a continuous surface without any T-junctions and cracks. Whether produced by simplification or refinement, for M to be valid it must satisfy the following property:

$$j \in M \Longrightarrow i \in M \qquad j \in C_i, \tag{1}$$

where C_i is the set of children of i. That is, for j to be active, its parents (and, by induction, all of its ancestors) must be active. Fig. 3 illustrates this property, where the dashed edges must be added to form a valid mesh. Even when the DAG traversal is top-down, ensuring this property is not as

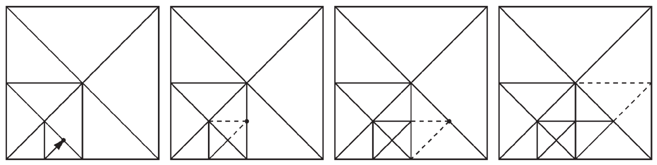


Fig. 3. Example of adaptively refined mesh. The bisection marked with an arrow sets off a cascading sequence of bisections (dashed lines) needed to avoid cracks (marked with dots) in the mesh. Our algorithm avoids this explicit patching of cracks.

easy as it may seem since it is possible to reach j without visiting both of its parents.

One solution to enforcing the validity of the mesh is to maintain explicit dependencies between each child and its parents [2]. However, this approach is inefficient both in terms of computation and storage. Our approach, instead, is to satisfy (1) by ensuring that the error terms used in the refinement criterion are nested, thereby implicitly forcing all parent vertices to be activated with or before their descendants.

3.2 Refinement Criterion

The idea of nested errors is not new. Pajarola [4] and Gerstner [11] discuss nested object space errors, and refer to the nesting condition as "saturating" the errors. However, neither describe how to guarantee that the errors after projection to screen space remain nested, which, as we shall see, requires that special care be taken in formulating the error metric. The ROAM algorithm [3] uses nested errors in both object and screen space. However, their screen space metric applies only to a restricted class of object space metrics and assumes that perspective projection is used. In addition, their metric appears considerably more expensive to evaluate than the ones presented here.

Perhaps the most closely related refinement algorithm to ours is the one by Blow [10]. His method, like ours, uses a nested sphere hierarchy. Each sphere is centered on the position $\mathbf{p}_i$ of a mesh vertex i and represents the isocontour of i's screen space error $\rho_i = \rho(\epsilon_i, \mathbf{p}_i, \mathbf{e})$, where ϵ_i is an object (or world) space error term for i and $\mathbf{e}$ is the viewpoint.[1] For a fixed screen space error tolerance τ, the isocontour for which $\rho_i = \tau$ divides space into two halves; i is active when the viewpoint is inside the sphere ($\rho_i > \tau$), and inactive for viewpoints outside it ($\rho_i < \tau$). Using these spherical isosurfaces, Blow constructs a forest of nested sphere hierarchies in which each parent sphere contains its child spheres. The vertices associated with these spheres need not be related in the refinement—as long as the viewpoint is outside a particular sphere, none of the vertices in the sphere's subtree can be active, which allows large groups of vertices to be eliminated quickly.

While theoretically simple, Blow's method has a number of drawbacks. First, to ensure nesting, τ must be fixed up front. Second, the method is tied to a particular error metric, a metric based on distance alone. A metric that varies with direction to the viewer, such as the one in [2], does not necessarily lead to isosurfaces with good nesting properties. Third, without maintaining explicit dependencies between vertices or inflating the spheres wherever necessary, (1) will generally not be satisfied, resulting in cracks. Finally, every tree in the sphere forest must be visited during refinement. Since this forest can be large, further clustering of the trees may be necessary.

Our approach bears some resemblance to Blow's, but avoids many of these shortcomings. We, too, use a DAG of nested spheres, but for a different purpose, and its structure is given by the vertex relationships in the refinement. In the discussion below, it is unimportant how the error terms ϵ and ρ are measured—we will discuss possible metrics later in Section 3.3. However, we require that $\rho(\epsilon, \mathbf{p}, \mathbf{e})$ increases monotonically with ϵ for fixed $\mathbf{p}$ and $\mathbf{e}$. This is a reasonable requirement; as ϵ increases, we would expect its projection ρ for a given viewpoint to increase as well (or at least remain the same). Using these definitions, a sufficient condition for satisfying (1) is:

$$\rho(\epsilon_i, \mathbf{p}_i, \mathbf{e}) \geq \rho(\epsilon_j, \mathbf{p}_j, \mathbf{e}) \qquad \forall j \in C_i.$$

This is the view-dependent version of the saturation condition in [31]. To guarantee this property, we could inflate the projected error for i by taking the maximum of ρ_i and ρ_j for all children j. However, we need this relationship to be transitive, i.e., it has to hold not only for i and its children, but also for all of i's descendants. Visiting every descendant of each active vertex at runtime is clearly impractical for large terrains. Instead, we compute a conservative bound on ρ_i by making use of our sphere hierarchy.

First, observe that ρ_i is made up of two distinct components: an object space error term ϵ_i and a view-dependent term that relates $\mathbf{p}_i$ and $\mathbf{e}$. Our approach is to separate the two and guarantee a nesting for each term. Let

$$\epsilon_i = \begin{cases} \hat{\epsilon}_i & \text{if } i \text{ is a leaf node} \\ \max\{\hat{\epsilon}_i, \max\limits_{j \in C_i}\{\epsilon_j\}\} & \text{otherwise,} \end{cases} \tag{2}$$

where $\hat{\epsilon}_i$ is the actual (not necessarily nested) geometric error measured by the object space metric. Then, clearly, $\epsilon_i \geq \epsilon_j$ for $j \in C_i$. Due to the monotonic relationship between ρ_i and ϵ_i, we must have $\rho(\epsilon_i, \mathbf{p}_i, \mathbf{e}) \geq \rho(\hat{\epsilon}_i, \mathbf{p}_i, \mathbf{e})$, which ensures that there is no loss in visual accuracy. We don't necessarily have $\rho(\epsilon_i, \mathbf{p}_i, \mathbf{e}) \geq \rho(\epsilon_j, \mathbf{p}_j, \mathbf{e})$ for $j \in C_i$, however, since an error projected from $\mathbf{p}_j$ may be arbitrarily larger than an error projected from $\mathbf{p}_i$ (e.g., the viewpoint may be close to $\mathbf{p}_j$ but far from $\mathbf{p}_i$). Therefore, it is not sufficient to nest the object space errors alone, but we must also account for this spatial relationship between parent and child vertices. This is accomplished by enclosing all descendants of i in a ball $B_i = \{\mathbf{x} : \|\mathbf{x} - \mathbf{p}_i\| \leq r_i\}$ centered on $\mathbf{p}_i$, with radius

$$r_i = \begin{cases} 0 & \text{if } i \text{ is a leaf node} \\ \max\limits_{j \in C_i}\{\|\mathbf{p}_i - \mathbf{p}_j\| + r_j\} & \text{otherwise.} \end{cases} \tag{3}$$

Then, $B_i \supseteq B_j$ for $j \in C_i$, i.e., the ball hierarchy is nested. A 2D example of this nesting is shown in Fig. 4. Finally, we define the maximum projected error as:

$$\rho_i = \rho(\epsilon_i, B_i, \mathbf{e}) = \max_{\mathbf{x} \in B_i} \rho(\epsilon_i, \mathbf{x}, \mathbf{e}). \tag{4}$$

Because $\epsilon_i \geq \epsilon_j$, $B_i \supseteq B_j$, and ρ is monotonic, we must have $\rho_i \geq \rho_j$ for $j \in C_i$. Consequently, if j is active, then so is its parent i, which is what we set out to show.

To compute ρ_i at runtime, we need to perform a constrained optimization over the ball B_i. Typically $\nabla\rho$ is nonzero everywhere (except possibly at the viewpoint) and the maximum of ρ occurs on the boundary of B_i.

1. In the remainder of this paper, we assume that the generic screen space error ρ_i is a function of the position of i and the viewpoint. Some error metrics may measure error at points other than the vertex positions (e.g., over entire triangles [3], [8]) and may depend on additional view information (e.g., gaze direction [3]). It should be straightforward to generalize our definitions to such error metrics.

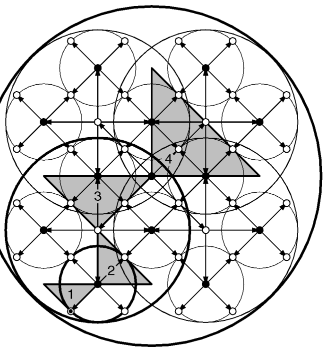


Fig. 4. Two-dimensional analogue of the nested sphere hierarchy used for refinement and view culling. The four triangles are associated with the vertices at their right-angle corners. Notice that the bounding spheres do not completely contain their corresponding triangles on the bottom two levels in the DAG, but do contain them on level 3 and above.

Nevertheless, finding this maximum may seem like an expensive process, although it turns out that it is generally easy to find a simple, closed form expression for the maximum and we will see in Section 3.3 how two different metrics can be expressed very concisely.

During preprocessing of the data set, we compute ϵ and r for each vertex using a bottom-up, level-by-level traversal of the vertex hierarchy. For a matrix-based layout, it is straightforward to both visit all vertices on a given level and identify their DAG children (see Fig. 2) in order to propagate ϵ and r up the DAG. For other data layouts, we make a second pass in which the vertices are rearranged by copying from the 2D matrix to the desired layout during simultaneous traversals of the two layouts in recursive refinement order (see Section 4 for more details).

In addition to the elevation z (and xy-coordinates in the domain, if so desired), ϵ and r are the only per-vertex parameters needed in our refinement algorithm. We again point out that we have so far left the choice of object space and screen space error metric entirely open. Given this general framework, we will now discuss how to compute actual screen space errors for different error metrics.

3.3 Error Metrics

In this section, we consider possible object space (ϵ) and screen space (ρ) error metrics. Typically, the screen space metric is defined in terms of a projection operator, i.e., ρ equals the projection of ϵ and it is often useful to treat the two metrics independently. Our framework is general enough to accommodate virtually any combination of error metrics, which will be illustrated in the following sections by a small set of examples.

3.3.1 Object Space Error Metrics

Perhaps the most common object space error measure for height fields is the vertical distance between corresponding points in the original and the approximating mesh. For simplicity, these errors are often computed at the height field vertices only [2], [9], but may be computed over triangles or even larger regions of influence associated with a vertex [3], [8]. Our framework accommodates both of these approaches since the position or region over which the object space error is measured can always be included in a vertex's (possibly inflated) bounding sphere.

Object space errors can also be measured incrementally, between two consecutive levels of refinement [2], or as the maximum error with respect to the full-resolution mesh [3], [8]. The incremental error is a good indicator of how much the mesh would *change* by removing a vertex, which may be a useful measure for estimating "popping" artifacts (see Section 3.6). The maximum error, on the other hand, is a bound on how far the mesh would deviate from the full-resolution surface if the vertex were removed. The choice between incremental and maximum errors is orthogonal to our refinement method and we will later present results of using both of these metrics.

Formally, we write the incremental and maximum vertical errors for a vertex i in terms of the triangles T_i in the diamond of i (Fig. 1). Let $z_t(x_i, y_i)$ be the elevation of triangle t at the point where vertex i lies in the domain, and let $\hat{\delta}_{i,t} = |z_i - z_t(x_i, y_i)|$ denote the vertical error between i and t. The incremental error can then be written as:

$$\hat{\epsilon}_i^{inc} = \max_{t \in T_i}\{\hat{\delta}_{i,t}\} = \left| z_i - \frac{z_l + z_r}{2} \right|. \quad (5)$$

That is, $\hat{\epsilon}_i^{inc}$ is the vertical displacement from i to the midpoint of its split edge $\{v_l, v_r\}$. The maximum error can similarly be written by considering i and its descendants:

$$\hat{\epsilon}_i^{max} = \max\left\{ \hat{\epsilon}_i^{inc}, \max_{t \in T_i} \max_{j \in D_{i,t}} \{\hat{\delta}_{j,t}\} \right\}, \quad (6)$$

where $D_{i,t}$ is the set of all descendants of i reached via recursive bisection of triangle t. Thus, the maximum error is the largest vertical distance from i and its descendants to the two (coarse) triangles in i's diamond. Note that the measured errors $\hat{\epsilon}_i^{inc}$ and $\hat{\epsilon}_i^{max}$ are not necessarily nested,[2] although $\hat{\epsilon}_i^{max}$ often is since it accounts for distances to all descendants of i. Finally, because the bounding sphere from (3) already contains $D_{i,t}$, we are ensured that the projection of $\hat{\epsilon}_i^{max}$ is a conservative error bound.

3.3.2 Isotropic Error Projection

Given an object space error ϵ, our view-dependent algorithm projects ϵ onto the screen to obtain a screen space error $\rho(\epsilon)$. While perspective projection is most commonly used to render the terrain, it involves problems with singularities and can be somewhat computationally inefficient. Therefore, it is common in view-dependent algorithms [2], [3], [8] to substitute the distance along the view direction with the Euclidean distance $d = \|\mathbf{e} - \mathbf{p}\|$ between the viewpoint $\mathbf{e}$ and the vertex position $\mathbf{p}$. The most simple metric of this form can be written as:

$$\rho(\epsilon, \mathbf{p}, \mathbf{e}) = \lambda \frac{\epsilon}{\|\mathbf{e} - \mathbf{p}\|} = \lambda \frac{\epsilon}{d}, \quad (7)$$

i.e., the projected error decreases with distance from the viewpoint. This is an *isotropic* error measure in the sense that ρ is the same in every direction a fixed distance d from the vertex. For the usual perspective projection onto a plane,

2. Indeed, it is possible for $\hat{\epsilon}^{max}$ to *increase* from one level to the next as a result of inserting a vertex.

$\lambda = \frac{w}{2\tan\varphi/2}$, where w is the number of pixels along the field of view φ. Equation (7) is in actuality a projection onto a sphere and not a plane, so a more appropriate choice is $\lambda = \frac{w}{\varphi}$. We then compare ρ against a user-specified screen space error tolerance τ.

In our refinement procedure, we need to find the maximum projection $\rho(\epsilon, B, \mathbf{e})$ over a set of points $\mathbf{x} \in B$. For the isotropic error, the maximum projection occurs where $\|\mathbf{e} - \mathbf{x}\|$ is minimized. For viewpoints inside B, this term is zero and we activate the vertex. If $\mathbf{e} \notin B$, then the minimum is $d - r$ and our maximum projected error becomes

$$\rho(\epsilon, B, \mathbf{e}) = \max_{\mathbf{x}\in B} \rho(\epsilon, \mathbf{x}, \mathbf{e}) = \lambda \frac{\epsilon}{d - r}. \tag{8}$$

Comparing ρ against τ and rearranging and squaring some terms (to avoid costly square roots), we obtain

$$\begin{aligned} active(i) &\Longleftrightarrow \rho(\epsilon_i, B_i, \mathbf{e}) > \tau \\ &\Longleftrightarrow \lambda \frac{\epsilon_i}{d_i - r_i} > \tau \\ &\Longleftrightarrow \frac{\lambda}{\tau}\epsilon_i > d_i - r_i \\ &\Longleftrightarrow (\nu\epsilon_i + r_i)^2 > d_i^2, \end{aligned} \tag{9}$$

where $\nu = \frac{\lambda}{\tau}$ is constant for each refinement pass. For spherical projection, $\kappa = \frac{1}{\nu}$ is the angular error threshold in radians. The above expression involves only six additions and five multiplications and is therefore very efficient to evaluate.

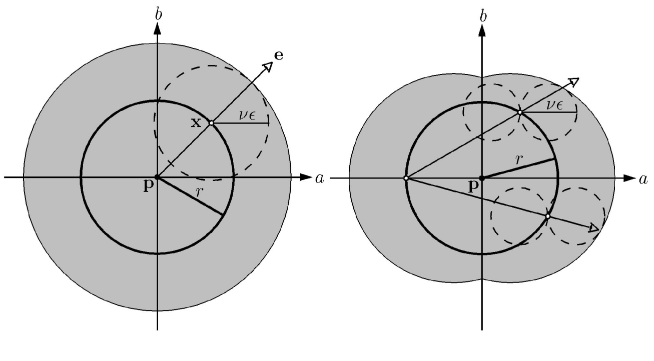
In a strict mathematical sense, (9) is valid only if our assumption $\mathbf{e} \notin B_i$ holds. However, if we use the convention that $\mathbf{e} \in B_i \Longrightarrow active(i)$, then (9) can correctly be used for all viewpoints, with one caveat: If $\epsilon_i = 0$, then its projection ought to also be zero, regardless of where the viewpoint is, and the vertex arguably should be deactivated. Of course, we could explicitly test for the special case $\epsilon_i = 0, \mathbf{e} \in B_i$ if it is considered important.

3.3.3 Anisotropic Error Projection

If object space errors are measured vertically, then errors viewed from above appear relatively smaller than errors viewed from the side. As a consequence, vertices directly below the viewer can often be eliminated. Lindstrom et al. [2] describe an *anisotropic* metric $\vec{\rho}$ that exploits this fact. While this metric leads only to marginally fewer triangles, we will here describe how to incorporate it into our framework for illustrative purposes. This metric fundamentally depends on the horizontal (xy) and vertical (z) components a and b, respectively, of d, and we can work with $\vec{\rho}$ in two dimensions to simplify matters. Using these conventions, the anisotropic metric can be written as:

$$\begin{aligned} \vec{\rho}(\epsilon, \mathbf{p}, \mathbf{e}) &= \lambda \frac{\epsilon\sqrt{(e_x - p_x)^2 + (e_y - p_y)^2}}{\|\mathbf{e} - \mathbf{p}\|^2} \\ &= \left(\lambda\frac{\epsilon}{d}\right)\left(\frac{a}{d}\right) \\ &= \rho(\epsilon, \mathbf{p}, \mathbf{e})\cos\theta, \end{aligned} \tag{10}$$

where θ is the angle of $\mathbf{e} - \mathbf{p}$ above the horizon. As $\mathbf{e}$ approaches directly above $\mathbf{p}$, θ approaches $\frac{\pi}{2}$ and the projected error vanishes. If, on the other hand, $\mathbf{e}$ is at the same elevation as $\mathbf{p}$, then θ is zero and $\vec{\rho}$ equals the isotropic error ρ. Fig. 5b shows the isocontours of $\vec{\rho}(\epsilon, \mathbf{x}, \mathbf{e})$ in 2D as being two abutting circles with diameter $\nu\epsilon$.



(a) (b)

Fig. 5. Two-dimensional illustration of isotropic and anisotropic error projection. The point $\mathbf{x}$ in the ball $B = (\mathbf{p}, r)$ is where the projection of the object space error ϵ is maximized. The dashed circles are isocontours $\rho(\epsilon, \mathbf{x}, \mathbf{e}) = \tau$ of the screen space error ρ for fixed ϵ and $\mathbf{x}$. The shaded regions show the sets of viewpoints $\mathbf{e}$ for which the vertex at $\mathbf{p}$ is active and equal the Minkowski sum of B and the isocontour interior. (a) Isotropic projection: $\mathbf{x}$ is the point closest to $\mathbf{e}$ or, equivalently, the intersection between the ball boundary and a line segment from $\mathbf{e}$ to $\mathbf{p}$. (b) Anisotropic projection: $\mathbf{x}$ is the intersection between the ball boundary and a line segment from $\mathbf{e}$ to the point opposite the *b*-axis where the ball meets the *a*-axis. The isocontours for two error maxima are shown.

We must now find the maximum $\vec{\rho}$ over all points $\mathbf{x}$ in B. It is relatively easy to show that if $\mathbf{e} \notin B$, then

$$\vec{\rho}(\epsilon, B, \mathbf{e}) = \max_{\mathbf{x}\in B} \vec{\rho}(\epsilon, \mathbf{x}, \mathbf{e}) = \lambda \frac{\epsilon}{d - r}\frac{a + r}{d + r}. \tag{11}$$

The maximum occurs on the boundary of B at a point shown in Fig. 5b. A simple activation condition associated with $\vec{\rho}_i$ can then be derived as follows:

$$\begin{aligned} active(i) &\Longleftrightarrow \vec{\rho}(\epsilon_i, B_i, \mathbf{e}) > \tau \\ &\Longleftrightarrow \lambda \frac{\epsilon_i}{d_i - r_i}\frac{a_i + r_i}{d_i + r_i} > \tau \\ &\Longleftrightarrow \epsilon_i(a_i + r_i) > \frac{\tau}{\lambda}(d_i^2 - r_i^2) \\ &\Longleftrightarrow \epsilon_i a_i > \kappa(d_i^2 - r_i^2) - \epsilon_i r_i \\ &\Longleftrightarrow \epsilon_i^2 a_i^2 > \max\{0, \kappa(d_i^2 - r_i^2) - \epsilon_i r_i\}^2, \end{aligned} \tag{12}$$

assuming $\epsilon_i > 0$ and $\mathbf{e} \notin B_i$. As in the isotropic case, however, this expression can be used unconditionally and, by reusing subexpressions, requires at most nine multiplications, seven additions, and one conditional branch.

3.4 Runtime Refinement

Having derived a criterion for selective refinement, we now summarize the algorithm for top-down, recursive refinement and on-the-fly triangle strip construction. Pseudocode for these steps is listed in Table 1. (We will see later how to incorporate view culling and geomorphing into this basic framework.) The refinement procedure builds a generalized triangle strip, $V = (v_0, v_1, v_2, \ldots, v_n)$, represented as a sequence of vertex indices.[3] A vertex v is appended to the

3. An *OpenGL* implementation would make repeated calls to `glVertex` with this sequence of vertices.

TABLE 1
Pseudocode for Recursive Mesh Refinement and Triangle Stripping

```
tstrip-append($V, v, p$)
1 if $v \neq v_{n-1}$ and $v \neq v_n$ then
2     if $p \neq parity(V)$ then
3         $parity(V) \leftarrow p$
4     else
5         $V \leftarrow (V, v_{n-1})$
6     $V \leftarrow (V, v)$

submesh-refine($V, i, j, l$)
1 if $l > 0$ and active($i$) then
2     submesh-refine($V, j, c_l(i, j), l - 1$)
3     tstrip-append($V, i, l \bmod 2$)
4     submesh-refine($V, j, c_r(i, j), l - 1$)

mesh-refine($V, n$)
1 $V \leftarrow (i_{sw}, i_{sw})$
2 $parity(V) \leftarrow 0$
3 for each $(j, k) \in ((i_s, i_{se}), (i_e, i_{ne}), (i_n, i_{nw}), (i_w, i_{sw}))$
4     submesh-refine($V, i_c, j, n$)
5     tstrip-append($V, k, 1$)
```

strip using the procedure tstrip-append. Line 5 "turns corners" in the triangulation by effectively swapping the two most recent vertices, which results in a degenerate triangle that is discarded by the graphics system [32]. Swapping is done to ensure that the parity—whether a vertex is on an even or odd refinement level—is alternating, which is necessary to form a valid triangle mesh. To this end, the two-state variable $parity(V)$ records the parity of the last vertex in V. Fig. 6 illustrates the sequence of triangles traversed during refinement.

The procedure submesh-refine corresponds to the innermost recursive traversal of the mesh hierarchy, where c_l and c_r are the left and right child vertices of the DAG parent j for the current triangle (Fig. 6). The designations "left" and "right" child do not necessarily correspond to making left and right turns when traversing the DAG. Instead, the sense of left and right alternates between consecutive levels. This is illustrated in Fig. 6, where we have labeled the left and right triangle children for a few levels in the binary triangle tree formed by the edge bisection. This hierarchy extends to DAG vertices by mapping diamonds (pairs of triangles) to their corresponding vertices (Fig. 1). For now, this geometric definition of c_l and c_r is sufficient. We will discuss how to compute these indices numerically from their ancestors i and j in Section 4.

Notice that submesh-refine in Table 1 is always called recursively with j as the new parent vertex, and the condition on line 1 is subsequently evaluated twice, once in each subtree. Therefore, most per-vertex work, such as testing for refinement, culling, morphing, etc., is unnecessarily duplicated. Because the evaluation of the refinement condition accounts for the majority of the refinement time, it is more efficient to move it up one level in the recursion, thereby evaluating it only once, and then conditionally making the recursive calls. For the sake of clarity, however, we will stick to the more concise way of writing our recursive functions throughout this paper.

Finally, the outermost procedure mesh-refine starts with a base mesh of four triangles (Fig. 2a) and calls submesh-refine once for each triangle. Here, n is the number of refinement levels, i_c the vertex at the center of the grid, $\{i_{sw}, i_{se}, i_{ne}, i_{nw}\}$ the four grid corners (Fig. 2a), and $\{i_n, i_e, i_s, i_w\}$ the vertices introduced in the first refinement step (Fig. 2b). The triangle strip is initialized with two copies of the same vertex to allow line 1 in tstrip-append to be evaluated. The first vertex, v_0, is then discarded after the triangle strip has been constructed.

For applications that demand the highest possible frame rates, it is common to parallelize the otherwise sequential, interleaved tasks of refinement and rendering as two asynchronous processes or threads [14], [18]. Our terrain visualization system supports this multithreaded model, where the render thread is periodically and asynchronously supplied with a list of geometry to render by the refinement thread.

3.5 View Frustum Culling

The rendering performance of our terrain visualization system is substantially improved by culling mesh triangles that fall outside the view volume. Our view culling, which is done as part of the recursive refinement, exploits the hierarchical nature of the subdivision mesh and culls large chunks of triangles high up in the mesh hierarchy whenever possible. Our approach is based on the culling algorithm outlined in [3], but is somewhat more efficient. In particular, we exploit the nested bounding sphere hierarchy to perform view culling, similar to [6], [33].

Note that the bounding sphere for a vertex i contains the vertices of all descendants of i. Thus, if the bounding sphere is not visible, then neither i nor its descendants will appear on the screen. It is possible, however, for a piece of a triangle t that has i or one of its descendants as a vertex to be visible, even though none of these vertices are visible. By excluding i, a coarser triangle than t will be rendered. To guarantee that such false positives in the culling test never occur, the bounding sphere for i could be expanded wherever necessary to contain i's incident triangles. Alternatively, the radius of a separate bounding sphere for view

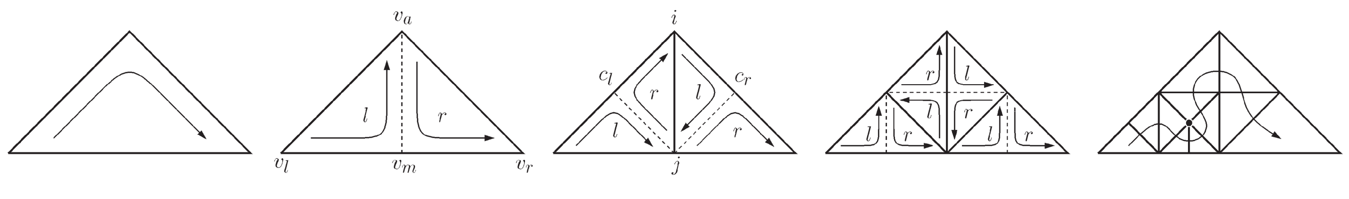


Fig. 6. Binary triangle tree formed by bisection. The labels l and r correspond to left and right children in the tree, respectively, while the arrows indicate the alternating triangle orientation on consecutive levels. The rightmost mesh shows that the recursive bintree traversal follows the Sierpinski curve and leads to a generalized triangle strip. The marked vertex illustrates a situation where swapping is needed in order to pivot around the vertex.

TABLE 2
Pseudocode for View Frustrum Culling

```
visible($i$, $inside$)
1 for each view frustum plane $\langle \hat{\mathbf{n}}_k, d_k \rangle$
2     if $\neg inside_k$ then
3         $s \leftarrow \hat{\mathbf{n}}_k \cdot \mathbf{p}_i + d_k$
4         if $s > r_i$ then
5             return false
6         if $s < -r_i$ then
7             $inside_k \leftarrow \mathbf{true}$
8 return true

submesh-refine-visible($V, i, j, l, inside$)
1 if $inside_k$ $\forall k$ then
2     submesh-refine($V, i, j, l$)
3 else if $l > 0$ and active($i$) and visible($i, inside$) then
4     submesh-refine-visible($V, j, c_l(i, j), l - 1, inside$)
5     tstrip-append($V, i, l \bmod 2$)
6     submesh-refine-visible($V, j, c_r(i, j), l - 1, inside$)
```

culling purposes could be stored with each vertex. In practice, however, the bounding sphere hierarchy is already loose enough that, at least in the domain, the incident triangles are contained above the second finest refinement level (see Fig. 4). Therefore, we have chosen to use the existing hierarchy for view culling, and have seen no visible artifacts of culling the mesh.

The pseudocode in Table 2 summarizes our view culling algorithm. The algorithm makes use of the six planes of the view frustum. The parameters $\langle \hat{\mathbf{n}}_k, d_k \rangle$ for each implicit plane equation $s = \hat{\mathbf{n}}_k \cdot \mathbf{x} + d_k = 0$ are computed in object space coordinates and are passed along in the refinement. We ensure that $\hat{\mathbf{n}}$ is a unit length vector so that s is the signed distance to the plane. As in [3], we maintain one flag for each plane, $inside_k$, indicating whether the bounding sphere is completely on the interior side of the plane with respect to the view volume. If this is the case, then all descendants' bounding spheres must also be on the interior side and no further culling tests against this plane are necessary. If the sphere is on the interior side of all six planes (line 2 of submesh-refine-visible), then we simply transition to our regular refinement procedure without view frustum culling. If, on the other hand, the sphere is entirely outside any one of the six planes, the vertex and its descendants are culled, and the refinement recursion terminates. Thus, view culling is done only for those spheres that straddle the planes of the view volume.

3.6 Geomorphing

Using our adaptive refinement and view frustum culling algorithms, we can generally maintain high frame rates at no perceptible loss in geometric quality. For example, we typically achieve 60 or more frames per second using a 640×480 window and a two-pixel tolerance. However, for high screen resolutions, the tolerance becomes relatively so small that the resulting adaptive meshes can easily exceed 100,000 triangles, too complex even for state-of-the-art graphics hardware to render at these display rates. Increasing the pixel tolerance mitigates this problem, but, when τ exceeds a few pixels, temporal artifacts become apparent. This is because each vertex insertion results in an instantaneous change in the geometry on the order of τ pixels and a noticeable and quite disturbing "pop" can be seen near the new vertex. This problem is exacerbated by using flat (per-triangle) shading, in which case, the temporal discontinuity in geometry, and, hence, in surface normals, results in a dramatic change in shading.

A well-known solution to this problem is to use *geomorphing* [12] to smooth out the transitions in geometry over several frames. This is generally done by defining the vertex position as a parametric function $\mathbf{p}(t)$: Whenever a new vertex is inserted, it is initially placed on the current surface ($t = 0$) and is, over time, slowly moved to its final position ($t = 1$). Conversely, removal of visible vertices is done by reversing the morphing process. For our terrain meshes, we interpolate only the elevation z:

$$z(t) = tz_i + (1 - t)\frac{z_l + z_r}{2} \qquad 0 \le t \le 1, \tag{13}$$

where z_i is the actual, measured elevation of vertex i and z_l and z_r are the elevations of the endpoints of i's split edge e_i (Fig. 1). Thus, when $t = 0$, i is at the midpoint of e_i and the geometry is locally no different from when i is absent. Note that z_l and z_r may be the elevations resulting from ongoing morphs for these two vertices and we may sometimes have a cascading sequence of morphs. Therefore, z_l and z_r must be passed along in the recursion. Fortunately, these vertices have already been visited and their elevations computed higher up in the recursion by the time we reach i.

There are two main approaches to geomorphing—time-based and position-based morphing—and they differ mainly in how the parameter t is defined. In time-based morphing [3], [8], the transition occurs over a fixed period of time or number of frames. This approach can be somewhat difficult to implement since it requires keeping track of morph start and/or end times for each vertex and imposes additional vertex dependencies, e.g., one may have to wait for a morph, or even a cascade of morphs, to finish before a vertex can be removed [8].

As its name suggests, position-based morphing [4], [9], [13], [15], [34] uses the position of the viewer instead of time to define the morph parameter t. For example, t could be a function of distance to the vertex [13]. One advantage of the position-based approach is that both the connectivity and geometry of the mesh depend only on the camera position, i.e., the mesh always looks the same from any given viewpoint $\mathbf{e}$. If, in addition, $t(\mathbf{e})$ varies smoothly with $\mathbf{e}$, then the mesh geometry is a continuous function of $\mathbf{e}$. Yet another advantage is that, in general, no state information is required to keep track of when the morph was initiated.

Because of its many desirable features, we have chosen to incorporate position-based morphing into our refinement algorithm. We approach this problem by defining a range of thresholds (τ_{min}, τ_{max}) and use morphing whenever the screen space error ρ, which is a function of the viewpoint, falls within this range. We show how this is done for the isotropic error metric from Section 3.3.2.

Our goal is to compute the morph parameter t. One possible approach would be to define t as:

$$t = \frac{\rho - \tau_{min}}{\tau_{max} - \tau_{min}} = \frac{\lambda \frac{\epsilon}{d - r} - \tau_{min}}{\tau_{max} - \tau_{min}}.$$

TABLE 3
Pseudocode for Geomorphing

```
morph($i$)
 1 $d \leftarrow \|\mathbf{e} - \mathbf{p}_i\|$
 2 $d_{max} \leftarrow \nu_{max}\epsilon_i + r_i$
 3 if $d < d_{max}$ then
 4     $d_{min} \leftarrow \nu_{min}\epsilon_i + r_i$
 5     if $d > d_{min}$ then
 6         return $\frac{d_{max}^2 - d^2}{d_{max}^2 - d_{min}^2}$
 7     else
 8         return 1
 9 else
10     return 0

submesh-morph($V, i, j, l, z_l, z_a, z_r$)
 1 if $l > 0$ and $(t \leftarrow$ morph($i$)$) > 0$ then
 2     $z \leftarrow tz_i + (1 - t)z_a$
 3     submesh-morph($V, j, c_l(i, j), l - 1, z_l, \frac{z_l + z_r}{2}, z$)
 4     tstrip-append-point($V, x_i, y_i, z, l \bmod 2$)
 5     submesh-morph($V, j, c_r(i, j), l - 1, z, \frac{z_l + z_r}{2}, z_r$)
```

Then, whenever $t \leq 0$, the vertex is inactive, while $t \geq 1$ implies that the vertex is active and fully morphed. This expression for t involves two divisions and a square root. We can find a simpler (although different) expression by making use of the activation condition in (9). That is, we define a range (d_{min}, d_{max}) for the distance d from the viewpoint to the vertex:

$$d_{min} = \frac{\lambda}{\tau_{max}}\epsilon + r = \nu_{min}\epsilon + r, \tag{14}$$

$$d_{max} = \frac{\lambda}{\tau_{min}}\epsilon + r = \nu_{max}\epsilon + r. \tag{15}$$

Then, since ρ and d are inversely proportional, $\rho = \tau_{min}$ whenever $d = d_{max}$ and $\rho = \tau_{max}$ whenever $d = d_{min}$. Finally, we define t as

$$t = \frac{d_{max}^2 - d^2}{d_{max}^2 - d_{min}^2}, \tag{16}$$

where we have squared the distances to avoid square roots. Thus, t, and, by extension, the vertex position, varies quadratically and smoothly with the vertex distance.

Table 3 summarizes our geomorphing algorithm for the isotropic metric. Because all distances computed are nonnegative, we can compute their squares directly on lines 1, 2, and 4 in morph. To further improve the performance, we can compute z directly instead of t to avoid any redundant computations when $t = 0$ and $t = 1$. Note that the triangle strip V is no longer a list of vertex indices. Rather, each vertex i in the strip is specified directly by its morphed coordinates $\langle x_i, y_i, z\rangle$.

Whereas the computations involved in performing geomorphing add to the refinement time, the improved temporal quality of the animation often allows a considerably larger pixel tolerance to be used, which results in far fewer triangles and an overall shorter refinement time. As observed by Hoppe [8] and others, errors as large as several pixels may go unnoticed if geomorphing is used to mask any temporal artifacts. As is evidenced by the accompanying video (see Section 5), a lower threshold τ_{min} as large as six pixels for a 640×480 window can be used.

The temporal quality of the morph generally depends on its duration: If the morph time is too short, then not enough temporal continuity is provided. If, on the other hand, the morph time is very long, then either τ_{min} must be small, resulting in a high-complexity mesh, or τ_{max} must be large, resulting in an inaccurate mesh and a large number of costly morph computations. We experimented with several choices of τ_{min} and τ_{max}, and found that $\tau_{max} = \frac{3}{2}\tau_{min}$ provided a good tradeoff between quality and complexity. We used this relationship for the animations and results in Section 5. The morph time also depends on the motion of the viewer. For high flight speeds, the morph time is short. On the other hand, the on-screen vertex motion due to a moving viewpoint often outweighs the motion due to morphing, which tends to reduce the effects of short morph times.

4 Data Layout and Indexing

In this section, we address the problem of laying out the terrain data on disk to achieve efficient out-of-core performance. In the spirit of our overall approach to terrain visualization, our goal is a simple but efficient mechanism for data paging. We achieve this by taking advantage of the paging mechanism of the operating system using the `mmap` system call,[4] which associates a part of the logical memory address space with a given disk file. Using this mechanism, the external memory part of our implementation consists simply of a single call to memory map the height field array of vertices stored on disk. After this step, the array is used as if it were allocated in main memory, while the operating system takes care of paging the data from disk as needed.

The main advantage of this approach is its simplicity. Moreover, since the paging mechanism is not specialized for a particular out-of-core algorithm, we can perform a fair comparison among different data layout schemes. In this paper, we study the performance potential intrinsic in different data layouts, without adding any specialized I/O layer with prefetching mechanisms that might further improve the out-of-core performance.

Given the framework described above, the external memory component can be reduced to a data layout problem. That is, we want to find a permutation of the set of mesh vertices such that their layout on disk closely follows the order in which they are typically accessed. We know the structure of the terrain traversal algorithm, and we have a transparent paging mechanism. Based on this, we need to determine: 1) a way of storing the raw data to minimize paging events, and 2) an efficient procedure for computing the index of a vertex in the given refinement order, with little or no computational overhead. The following two subsections describe a data layout scheme that satisfies requirements 1) and 2) and that has a particularly straightforward implementation.

4.1 Interleaved Quadtrees

Based on our refinement algorithm, each vertex (apart from the four corners of the grid) can be labeled as white, if introduced at an even level of refinement, or black, if introduced at an odd level. Fig. 2 shows this classification

4. The equivalent Windows function is called `MapViewOfFile`.

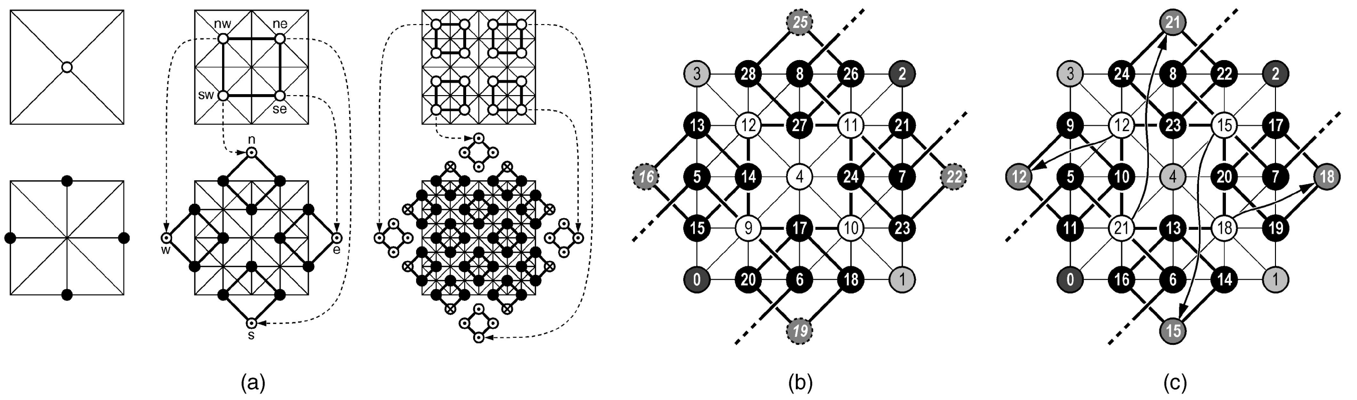


Fig. 7. (a) First three levels of the white (top) and black (bottom) quadtrees. A complete black quadtree is obtained by adding the dotted and crossed ghost vertices. The arrows indicate how to embed the white quadtree in the unused parts of the black quadtree. (b) Indices for the first few levels of the interleaved quadtrees (without embedding). (c) Indices for the embedded quadtrees.

for the first four levels of refinement. Fig. 7a shows how the white vertices form a quadtree—the *white quadtree*, Q_w. Interestingly, the black vertices can also be considered as part of a *black quadtree*, Q_b. Fig. 7a shows as dotted and crossed circles the vertices added outside the rectilinear grid to form a complete black quadtree. We will refer to these additional vertices as "ghost vertices." Note that Q_b does not start at the root but at the first level of refinement.

Since the traversal of the DAG (see Section 3.1) is performed top-down, starting from the root, good locality can be achieved by storing the data from coarse to fine levels. Within each level, the data should be stored so as to preserve neighborhood properties to the extent possible; geometrically close vertices should be stored close together in memory. For a quadtree, this can be achieved by using the order induced by the following formula for the index $c(p, k)$ of the kth child of the parent node p:

$$c(p, k) = 4p + k + m \qquad k = 0, 1, 2, 3, \tag{17}$$

where m is a constant dependent on the index of the root and the index distance between consecutive levels. Using this layout, all the vertices on the same level are stored together. The index distance between two vertices on a level depends on the distance to their common ancestor in the quadtree, e.g., any four siblings are stored in consecutive positions. For this indexing scheme, we interleave the black and white quadtrees, with roots $r_b = 3$ and $r_w = 4$. Since r_b is not used in practice, we assign the first four indices (0-3) to the corners of the grid. The first child of r_b is stored immediately after r_w, and we have

$$c(r_b, 0) = 4 \cdot 3 + 0 + m = 5,$$
$$c(r_w, 0) = 4 \cdot 4 + 0 + m = 9,$$

which both imply $m = -7$. Fig. 7b shows the vertex indices for the first few levels of the interleaved quadtrees.

4.2 Embedded Quadtrees

Notice in Fig. 7a that the ghost vertices in Q_b are not used. Because the data is eventually stored as a single linear array without any pointers between the quadtree nodes, this results in unwanted "holes" in the array. It is, however, possible to reduce the amount of unused space. First, observe that the total number of ghost vertices is roughly twice as large as the number of white vertices. As a consequence, instead of using two interleaved quadtrees, we can use the black quadtree only and store the white nodes in place of (a subset of) the ghost nodes. We divide Q_w into four subtrees, rooted at the children of r_w. Fig. 7a shows the insertion of these subtrees into the unused space of Q_b. The use of a single quadtree also affects the value of the constant m. In this case, we have $r_b = 4$ (this value is actually used for the white root) and $c(r_b, 0) = 5$, which implies $m = -11$. In addition, since the white quadtree has been split up into four independent subtrees, these relocated subtrees will not be reached from the white root r_w (node 4 in Fig. 7c) using (17). Therefore, we cannot begin the recursive refinement with r_w, but must unroll the recursion one level and make eight instead of four calls to submesh-refine from mesh-refine.

4.3 Efficient Index Computation

To avoid any overhead in the refinement process, we need an efficient method for computing the indices of the vertices visited in our top-down traversal of the terrain. For data stored in linear order (standard row major matrix layout), computing the child indices in the DAG can be made easy by carrying along three indices in the refinement: (v_l, v_a, v_r). These indices make up the current triangle t in the refinement and their subscripts correspond to the left, apex, and right corner of the triangle (Fig. 6). The two child triangles of t in the recursion can then be written as $t_l = (v_l, v_m, v_a)$ and $t_r = (v_a, v_m, v_r)$. Here, v_m corresponds to the vertex at the midpoint of the split edge $\{v_l, v_r\}$, which can be computed simply as the index average $v_m = (v_l + v_r)/2$.

For the indexing scheme based on the interleaved quadtrees, we make use of the parent-child relationship between vertices in the quadtrees. Consider one refinement step as shown in Fig. 6. The new white vertices c_l (left child) and c_r (right child) have a common *graph parent* p_g in the refinement DAG (j in the figure). Moreover, the graph parent of p_g is also the *quadtree parent* p_q of c_l and c_r (i in the figure). Based on this observation, the indices c_l and c_r can be computed from the index of their quadtree parent p_q using (17). The relative positions of p_q and p_g determine which two branches (the values of k) to use to reach c_l and

TABLE 4
Transition Table for Quadtree Branches

k_l, k_r		k_g							
		n·sw·0		e·se·1		s·ne·2		w·nw·3	
k_q	sw·n·0	se·e·1	ne·s·2	ne·s·2	nw·w·3	nw·w·3	sw·n·0	sw·n·0	se·e·1
	se·e·1	nw·w·3	sw·n·0	sw·n·0	se·e·1	se·e·1	ne·s·2	ne·s·2	nw·w·3
	ne·s·2	se·e·1	ne·s·2	ne·s·2	nw·w·3	nw·w·3	sw·n·0	sw·n·0	se·e·1
	nw·w·3	nw·w·3	sw·n·0	sw·n·0	se·e·1	se·e·1	ne·s·2	ne·s·2	nw·w·3

Table used to determine the left (k_l) and right (k_r) branch to be used in (17) for quadtree parent branch k_q and graph parent branch k_g. For example, $k_q = sw$, $k_g = n$ results in $k_l = se$, $k_r = ne$.

c_r from p_q. This relationship is summarized in Table 4. From (17), $k = (c - m) \bmod 4$ gives the quadtree branch k of p corresponding to c. That is, the value of k can be determined from the lowest two bits of the vertex index. We can then use (17) and Table 4 to compute c_l and c_r. However, there is considerable redundancy in the transition table and, by carefully numbering the four branches in the two quadtrees, it is possible to compute c_l and c_r using simple arithmetic. We show how this is done below.

In order to make the transition table as simple as possible, we have chosen to number the quadtree branches as follows: $sw = n = 0$, $se = e = 1$, $ne = s = 2$, $nw = w = 3$. This choice allows us to have a single table (and not two) for transitions between the two quadtrees. Let us focus on how to encode Table 4. We will use $k(k_q, k_g, b)$ to denote both k_l and k_r, with the convention that b is zero for k_l and one for k_r. First, observe that rows 0 and ~2 are the same, as are rows 1 and 3, thus

$$k(k_q, k_g, b) = k(k_q \bmod 2, k_g, b).$$

We can therefore focus only on rows 0 and 1. Now, notice that row 1 equals row 0 shifted twice to the right, i.e.,

$$k(k_q, k_g, b) = k\big(0, (2(k_q \bmod 2) + k_g) \bmod 4, b\big).$$

We are now left with row 0 only, from which we notice that $k_l = (k_q + 1) \bmod 4$ and $k_r = (k_l + 1) \bmod 4$, i.e.,

$$\begin{aligned} k(k_q, k_g, b) &= (2(k_q \bmod 2) + k_g + b + 1) \bmod 4 \\ &= (2k_q + k_g + b + 1) \bmod 4. \end{aligned}$$

Since $k_q = (p_q - m) \bmod 4$ and $k_g = (p_g - m) \bmod 4$,

$$\begin{aligned} k_l(p_q, p_g) &= (2(p_q - m) + (p_g - m) + 1) \bmod 4 \\ &= (2p_q + p_g - 3m + 1) \bmod 4 \\ &= (2p_q + p_g + m + 1) \bmod 4 \\ k_r(p_q, p_g) &= (2p_q + p_g + m + 2) \bmod 4. \end{aligned}$$

Finally, we obtain the following expressions for c_l and c_r:

$$c_l(p_q, p_g) = 4p_q + ((2p_q + p_g + m + 1) \bmod 4) + m, \tag{18}$$

$$c_r(p_q, p_g) = 4p_q + ((2p_q + p_g + m + 2) \bmod 4) + m. \tag{19}$$

These simple equations are used in the submesh-refine procedure in Section 3.4.

4.4 Memory-Efficient Hierarchical Indexing

One drawback of our quadtree-based indexing schemes is that they use a noncontiguous address space. In the case of interleaved quadtrees, the unused ghost vertices result in a 66 percent increase in storage, which is reduced to 33 percent by embedding the quadtrees. This overhead can be completely eliminated by using a layout based on a hierarchical version of the Π-order space filling curve.[5] Because the implementation of this scheme is not as straightforward as the quadtree-based schemes presented above, we will not describe it here, but refer the interested reader to [35]. In Section 5, we include empirical results of the performance achieved both with the quadtree-based schemes and with the hierarchical Π-order curve.

5 Results

In this section, we present the results of running an implementation of our terrain visualization system on several computer architectures. We used a two-processor 800 MHz Pentium III PC, running Red Hat Linux, with 900 MB of RAM and GeForce2 graphics. To push the out-of-core aspect of our system, we artificially limited the memory configuration of this machine to 64 MB for some of our results. A two-processor 300 MHz R12000 SGI Octane with Solid Impact graphics and 900 MB of RAM was also used to measure memory coherency, while we used a 48-processor 250 MHz R10000 SGI Onyx2 with 15.5 GB of RAM and InfiniteReality2 graphics to avoid being graphics and memory limited and to allow the raw refinement speed to be measured. For all results, we used a data set over the Puget Sound area in Washington (see Fig. 8), which is made up of $16,385 \times 16,385$ vertices at 10 meter horizontal and 0.1 meter vertical resolution.[6] Using our data structures, this data set occupies roughly 5 GB on disk. The quantitative results presented here were collected during a 2,816-frame fly-over of this data set. The window size was, in all cases, 640×480 pixels.

5.1 View-Dependent Refinement

We will first discuss the performance of our view-dependent refinement algorithm. We used the distance-based error metric described in Section 3.3.2 for the results presented here. To evaluate the efficiency in mesh complexity for a given accuracy, we recorded, for the fly-over, the number of rendered triangles obtained using both a bottom-up simplification of the terrain and our conservative top-down scheme. In the bottom-up scheme, the object space errors need not be inflated to guarantee nesting nor do the projected errors have to be inflated by measuring them over the nested bounding spheres. Instead, we used the actual projected error $\rho(\hat{\epsilon}_i, \mathbf{p}_i, \mathbf{e}) \leq \rho(\epsilon_i, B_i, \mathbf{e})$ of the measured error $\hat{\epsilon}_i$ for each vertex and then performed a separate step to patch cracks by activating the ancestors of all active vertices. Using this bottom-up approach, we obtain, for any given metric and tolerance, the minimal valid mesh possible,[7] which consequently serves as a good benchmark for evaluating our refinement method. Note that, contrary

5. The Π-order curve is similar to the Z-order curve described in [1].

6. This data set can be downloaded from http://www.cc.gatech.edu/projects/large_models/ps.html.

7. As noted earlier, it is possible that removing a vertex leads to a reduction in $\hat{\epsilon}$ (e.g., resulting in a mesh with a better fit) and, thus, possibly a reduction in ρ from above to below τ. For consistency and simplicity, we have chosen to be conservative and not allow such operations.

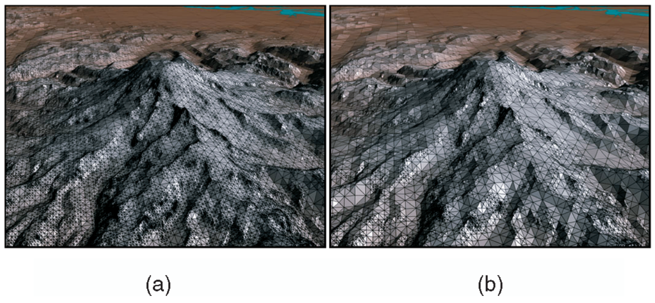


Fig. 8. View of Mount Rainier, Washington, illustrating subdivision meshes for two different screen space error thresholds τ. (a) $\tau = 2$; 79,382 triangles. (b) $\tau = 4$; 25,100 triangles.

to [1], we used view culling in our comparison between simplification and refinement.

Fig. 9 shows, for both the incremental ($\hat{\epsilon}^{inc}$) and maximum ($\hat{\epsilon}^{max}$) object space metrics, the number of triangles produced by bottom-up simplification, as well as the relative percentages of additional triangles produced by our conservative top-down refinement. We used a tolerance $\tau = 1$ pixel and a coarsened version of the Puget Sound data downsampled to $1,025 \times 1,025$ vertices (to make data collection tractable). As can be seen from the shaded regions, the maximum error $\hat{\epsilon}^{max}$ consistently resulted in a small increase in mesh complexity over using $\hat{\epsilon}^{inc}$. This is not surprising since $\hat{\epsilon}^{max} \geq \hat{\epsilon}^{inc}$ (Section 3.3.1). However, Fig. 9 shows that the discrepancy is often small, suggesting that $\hat{\epsilon}^{inc}$ can be used to approximate the more compute-intensive $\hat{\epsilon}^{max}$. Because of its simplicity, we used the incremental error for the remaining results below.

The two curves in Fig. 9 show on the rightmost axis the increase in number of triangles for top-down refinement relative to bottom-up simplification. Notice that, when the triangle counts are high (frames 1,024-2,048), the increase is about 1-2 percent for both metrics. The overhead is large only when the triangle counts are low, suggesting that our refinement produces a small, roughly constant increase in mesh complexity. However, because the large peaks occur only at low triangle counts, the net increase in complexity remains low. Over the entire fly-over, the total number of

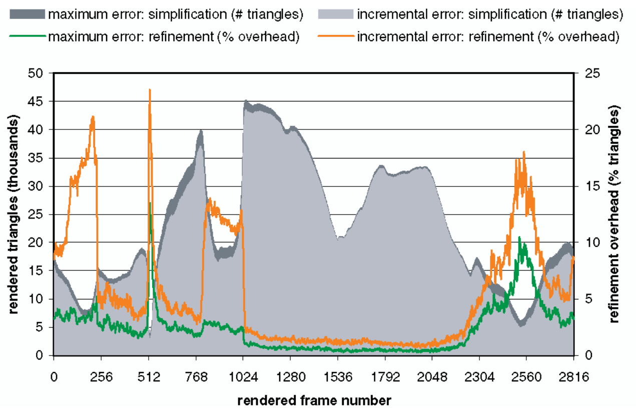


Fig. 9. Mesh complexity comparison between top-down refinement and optimal, bottom-up simplification. The graph shows, for two different object space metrics, the minimal number of triangles (shaded areas) and relative overhead (solid lines) due to refinement.

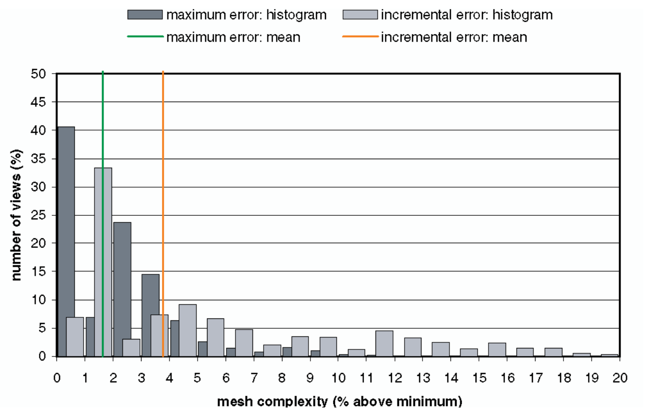


Fig. 10. Distribution and average of overhead in mesh complexity due to conservative refinement relative to the minimal mesh.

rendered triangles increased only by 1.63 percent and 3.76 percent for $\hat{\epsilon}^{max}$ and $\hat{\epsilon}^{inc}$, respectively. These averages and the corresponding highly skewed distributions are shown in Fig. 10. Because $\hat{\epsilon}^{max}$ is already close to nested, using simplification instead of refinement has less potential for improvement than in the case of incremental errors.

For the same flight path as above, over the full-resolution data set, we also measured the mesh complexity for the isotropic (ρ) and anisotropic ($\vec{\rho}$) screen space error metrics, using $\hat{\epsilon}^{inc}$ as the object space metric. We found that $\vec{\rho}$ on average led to a 2.5 percent reduction in mesh complexity over ρ, although at the expense of efficiency of evaluation. This behavior, observed also in [8], leads us to conclude that the simple isotropic metric is to be favored.

We next evaluate the performance increase due to culling and multithreading (one thread each for rendering and refinement). These results, which are summarized in Table 5 and plotted in Fig. 11, demonstrate a clear advantage of using both culling and multithreading. We were able to sustain 40,000 rendered triangles at 60 frames per second on the SGI Onyx2 during the fly-over. Above this triangle count, the frame rate slowed briefly. We generally obtained even higher, although more varied, frame rates on the PC (over 72 Hz on average). In both cases, we synchronized our rendering rate with the monitor display rate (60 Hz on the SGI, 75 Hz on the PC), which often resulted in significant idle time. This idle time is part of the frame time and overall flight time reported in Table 5.

Fig. 11b highlights the refinement performance, with and without culling, measured in the number of rendered triangles divided by the wall clock refinement time. For low triangle counts, the refinement runs faster when view culling is disabled, as expected. Notice, however, that, as the mesh complexity increases toward the middle of the graph, the lack of view culling leads to a significant decrease in performance. Conversely, the use of view culling results in a relative speedup. We attribute this result to caching behavior—as the triangle strip grows, an increasing number of cache misses are made, which slows down the method that did not use culling. Meanwhile, when a large fraction of triangles are culled, the overhead of making recursive function calls dominates, as evidenced by the sharp drop in performance near frame #512.

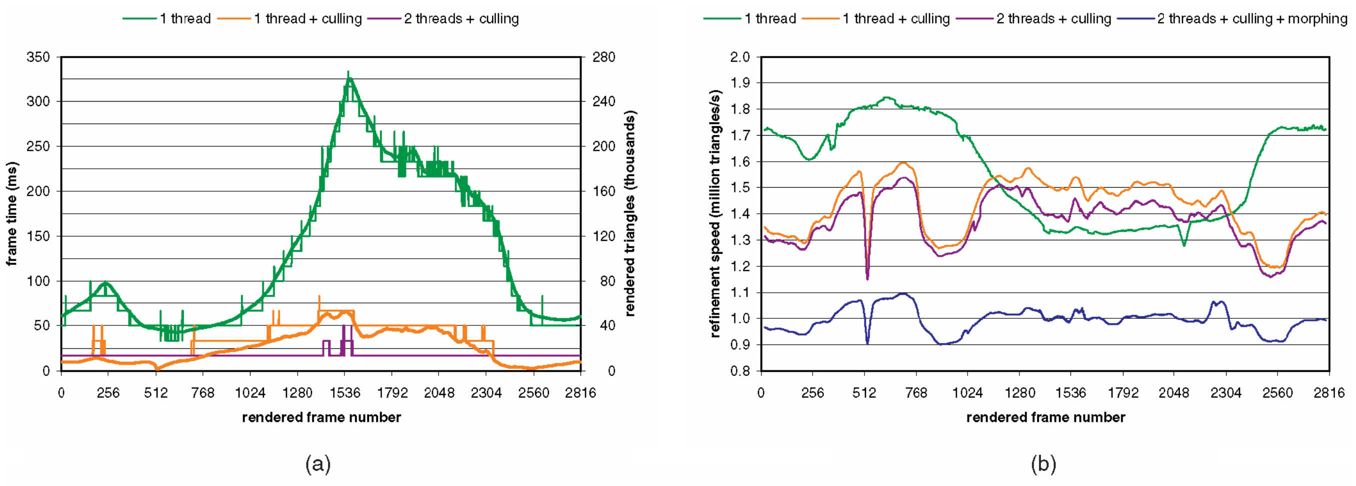


Fig. 11. In-core rendering and refinement performance on SGI Onyx2. The curves correspond to single-threading—with and without culling—and multithreading with culling—with and without geomorphing. The hierarchical indexing scheme was used in all four runs. (a) Frame time (thin lines) and rendered triangles (thick lines). Using multithreading, a steady 60 Hz is maintained during nearly the entire fly-over. The number of triangles for the three schemes that used culling coincide, therefore the graph for only one of them is shown. Similarly, the frame rates with and without geomorphing are roughly the same. (b) Refinement performance over time. The vertical axis corresponds to the number of nondegenerate triangles in the triangle strip divided by the (wall clock) refinement/culling/geomorphing time.

In order to show the qualitative performance of our geomorphs, we have made a number of MPEG animations available at http://www.cc.gatech.edu/~lindstro/papers/tvcg2002/. These animations show that, by using geomorphing, the screen space error tolerance can be increased considerably without any appreciable loss in quality. Objectionable temporal artifacts like geometric popping and shading discontinuities are virtually eliminated by morphing. Without morphing, such artifacts can be visually disturbing if the tolerance is larger than two pixels, while, using geomorphing, the lower error tolerance τ_{min} can be doubled or even tripled before the smooth motion caused by the geomorphs can even be detected.

Table 5 and Fig. 11b show some quantitative results of using geomorphing. Using $\tau_{max} = \frac{3}{2}\tau_{min}$ and a variety of choices for τ_{min}, we found that roughly 20-25 percent of the vertices were morphing at any one time. As expected, the additional work required to continuously morph the mesh geometry results in a drop in the refinement performance. However, as our animations show, the increase in acceptable tolerance due to improvements in temporal quality more than offsets the comparatively small increase in refinement time.

Finally, we evaluated the efficiency of using a single triangle strip and found that the strips averaged 1.56 vertices per nondegenerate triangle, which compares favorably to three vertices/triangle for independent triangles.

TABLE 5
Performance Results

platform	*geo-morphing*	*multi-threading*	*view culling*	*time (s)*	*rendering (Mtri/s)*	*refinement (Mtri/s)*
SGI (15.5 GB)				317.43	0.939	1.435
			✓	75.50	0.595	1.466
		✓	✓	47.63	0.944	1.396
	✓	✓	✓	47.71	0.942	0.990
PC (900 MB)				170.70	1.683	2.350
			✓	43.89	0.980	1.860
		✓	✓	38.62	1.113	1.777
	✓	✓	✓	38.98	1.103	1.515

Flight time and average performance for 2,816-frame fly-over (see also Fig. 11). The rendering performance is measured as the number of rendered triangles over the frame time (in multiples of the monitor frame time), which includes the refinement time in single-threaded mode.

5.2 Data Layout

We end this section by comparing the memory performance of four different indexing schemes: the embedded quadtree scheme from Section 4.2, the Π-order scheme, a blocking scheme based on 32×32 tiles from the full-resolution data, and a standard matrix layout in row major form. For all these methods we stored, with each vertex a 20-byte floating point record $\langle \mathbf{p}, \epsilon, r \rangle$.[8]

Fig. 12a shows for the Puget Sound fly-over the total number of page faults for varying values of the error tolerance τ. Smaller values of τ result in larger meshes and, therefore, more data being paged in. Clearly, our hierarchical indexing schemes greatly outperformed the linear and block-based schemes and often led to drastically improved paging speeds (Fig. 13). Perhaps surprisingly, the block-based scheme, which is often used for terrains, performs the worst of them all. This is because the refined mesh rarely consists of large groups of vertices at the highest resolution. Instead, a handful of vertices are needed from each block, requiring virtually the entire terrain to be paged in during each refinement pass. A more reasonable block-based indexing scheme would be to use subsampling to create a multiresolution pyramid for more coherent access to different resolutions of data. However, such an indexing scheme uses multiple indices per vertex, which would arguably make for an unfair comparison with our other indexing schemes.

8. We are not concerned here with the size of these records, which admittedly are unnecessarily large, but focus instead on how efficient the different indexing schemes are at accessing them.

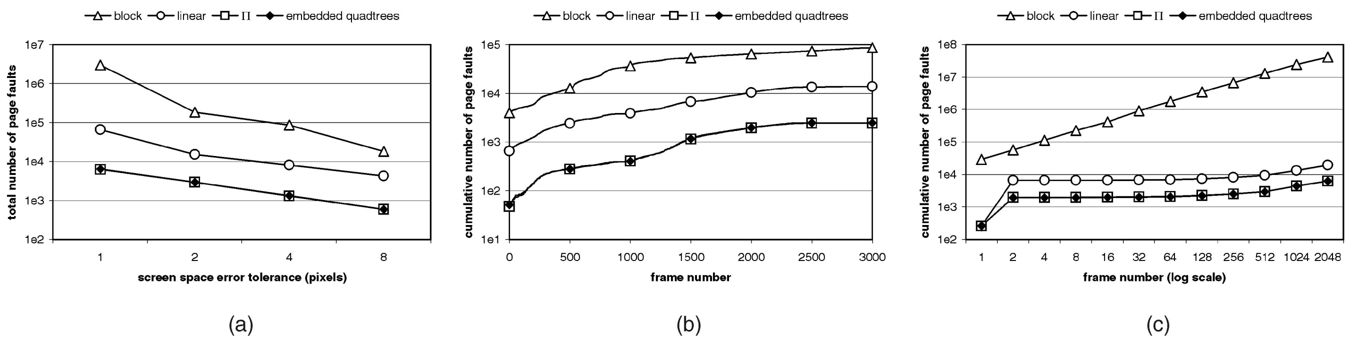


Fig. 12. (a) Total number of page faults vs. error tolerance τ. (b), (c) Cumulative number of page faults over time on two different platforms.

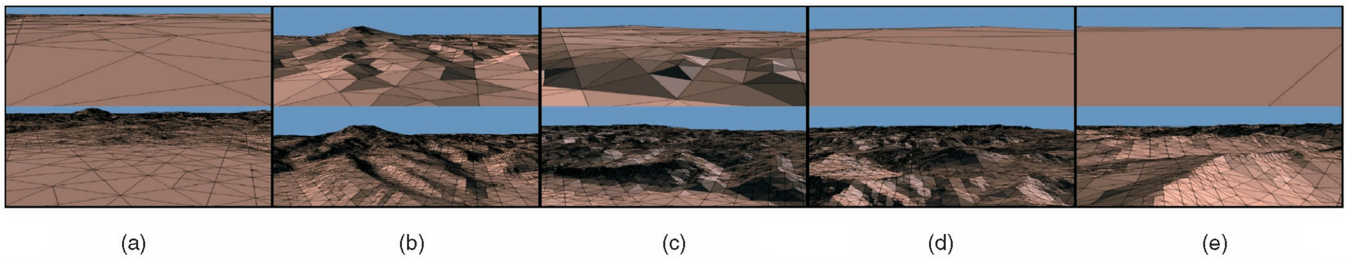


Fig. 13. Frames from two multithreaded fly-over sequences using linear, row major indexing (top) and quadtree-based indexing (bottom). The flight paths for the two sequences are the same. Detailed terrain is paged in faster using the quadtree-based scheme due to improved cache performance.

We also investigated the paging behavior over time. The graphs in Fig. 12b and Fig. 12c show that there is a significant hit at startup, when no data is memory resident, after which pages often stay in user memory or can be reclaimed quickly from the operating system's cache. These results also indicate that the hierarchical indexing schemes consistently result in one to two orders of magnitude improvement in paging performance over the nonhierarchical layouts.

Finally, we measured the raw in-core refinement speed of all indexing schemes. Due to better cache locality, the embedded quadtree scheme, while involving a few more operations, is still twice as fast as the linear scheme and is also twice as fast as the more complex Π-order scheme. This suggests that the linear scheme is inferior in all aspects to quadtree-based indexing, with the exception of memory overhead. We plan to investigate alternative indexing schemes that have the same desirable properties as the quadtree scheme, but with higher memory efficiency.

6 Summary

We have presented algorithms for two important components of large-scale terrain rendering: a method for efficient view-dependent refinement and an indexing scheme for organizing the data in a memory-friendly manner. The refinement and rendering components of our algorithm have been shown to be very efficient and, in spite of their simplicity, competitive with the state of the art in terrain visualization. Indeed, the core components of our view-dependent refinement and hierarchical indexing can be implemented in as little as a few dozen lines of C code. An implementation of our algorithms can be downloaded from http://www.cc.gatech.edu/~lindstro/software/soar/.

We will conclude by discussing some possible directions for future work. Whereas recent terrain visualization work has focused on view-dependent geometric approximation, perhaps an even more important component is texture LOD management. Few techniques currently exist for transparently caching and loading multiresolution textures. Another issue related to view-dependent methods, and, in particular, those supporting geomorphing, is how to efficiently render the continuously changing mesh. Current graphics hardware is not optimized for "immediate mode" rendering and, thus, hybrid methods that cache larger static pieces of the mesh may provide more efficient rendering. Finally, we envision that our algorithms for 2D height fields can be generalized to higher dimensions. For example, we intend to investigate how to extend our framework to view-dependent rendering of 3D scalar fields to support progressive isocontouring.

Acknowledgments

This work was performed under the auspices of the US Department of Energy by Lawrence Livermore National Laboratory under contract no. W-7405-Eng-48.

References

[1] P. Lindstrom and V. Pascucci, "Visualization of Large Terrains Made Easy," *Proc. IEEE Visualization 2001*, pp. 363-370, Oct. 2001.
[2] P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hodges, N. Faust, and G. Turner, "Real-Time, Continuous Level of Detail Rendering of Height Fields," *Proc. SIGGRAPH '96*, pp. 109-118, Aug. 1996.
[3] M.A. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineev-Weinstein, "ROAMing Terrain: Real-Time Optimally Adapting Meshes," *Proc. IEEE Visualization '97*, pp. 81-88, Nov. 1997.

[4] R.B. Pajarola, “Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation,” *Proc. IEEE Visualization ’98,* pp. 19-26, Oct. 1998.
[5] W. Evans, D. Kirkpatrick, and G. Townsend, “Right-Triangulated Irregular Networks,” *Algorithmica,* vol. 30, pp. 264-286, Mar. 2001.
[6] H. Hoppe, “View-Dependent Refinement of Progressive Meshes,” *Proc. SIGGRAPH ’97,* pp. 189-198, Aug. 1997.
[7] M. Gross, R. Gatti, and O. Staadt, “Fast Multiresolution Surface Meshing,” *Proc. IEEE Visualization ’95,* pp. 135-142, Oct. 1995.
[8] H. Hoppe, “Smooth View-Dependent Level-of-Detail Control and Its Application to Terrain Rendering,” *Proc. IEEE Visualization ’98,* pp. 35-42, Oct. 1998.
[9] S. Röttger, W. Heidrich, P. Slussallek, and H.-P. Seidel, “Real-Time Generation of Continuous Levels of Detail for Height Fields,” *Proc. Sixth Int’l Conf. Central Europe Computer Graphics and Visualization,* pp. 315-322, Feb. 1998.
[10] J. Blow, “Terrain Rendering at High Levels of Detail,” *Proc. 2000 Game Developers Conf.,* Mar. 2000.
[11] T. Gerstner, “Multiresolution Visualization and Compression of Global Topographic Data,” *Geoinformatica,* to appear. Available as SFB 256 Report 29, Univ. of Bonn, 1999.
[12] R.L. Ferguson, R. Economy, W.A. Kelly, and P.P. Ramos, “Continuous Terrain Level of Detail for Visual Simulation,” *Proc. 1990 IMAGE V Conf.,* pp. 144-151, June 1990.
[13] D. Cohen-Or and Y. Levanoni, “Temporal Continuity of Levels of Detail in Delaunay Triangulated Terrain,” *Proc. IEEE Visualization ’96,* pp. 37-42, Oct. 1996.
[14] J. Rohlf and J. Helman, “IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics,” *Proc. SIGGRAPH ’94,* pp. 381-395, July 1994.
[15] D. Cline and P.K. Egbert, “Terrain Decimation through Quadtree Morphing,” *IEEE Trans. Visualization and Computer Graphics,* vol. 7, pp. 62-69, Jan.-Mar. 2001.
[16] J.S. Vitter, “External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA,” *ACM Computing Surveys,* vol. 33, pp. 209-271, June 2001.
[17] D. Davis, T.Y. Jiang, W. Ribarsky, and N. Faust, “Intent, Perception, and Out-of-Core Visualization Applied to Terrain,” *Proc. IEEE Visualization ’98,* pp. 455-458, Oct. 1998.
[18] P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hodges, A.O. den Bosch, and N. Faust, “An Integrated Global GIS and Visual Simulation System,” Technical Report GIT-GVU-97-07, Georgia Inst. of Technology, Mar. 1997.
[19] M. Reddy, Y. Leclerc, L. Iverson, and N. Bletter, “TerraVision II: Visualizing Massive Terrain Databases in VRML,” *IEEE Computer Graphics and Applications,* vol. 19, no. 2, pp. 30-38, Mar./Apr. 1999.
[20] C. Gotsman and B. Rabinovitch, “Visualization of Large Terrains in Resource-Limited Computing Environments,” *Proc. IEEE Visualization ’97,* pp. 95-102, Nov. 1997.
[21] J. Döllner, K. Baumann, and K. Hinrichs, “Texturing Techniques for Terrain Visualization,” *Proc. IEEE Visualization 2000,* pp. 207-234, Oct. 2000.
[22] L. Balmelli, “Rate-Distortion Optimal Mesh Simplification for Communications,” PhD thesis, Ecole Polytechnique Federale de Lausanne, Switzerland, 2000.
[23] H. Samet, *Applications of Spatial Data Structures.* Reading, Mass.: Addison-Wesley, 1990.
[24] D.S. Wise, “Ahnentafel Indexing into Morton-Ordered Arrays, or Matrix Locality for Free,” *Proc. EUROPAR 2000: Parallel Processing,* pp. 774-784, Aug. 2000.
[25] V. Pascucci, “Multi-Resolution Indexing for Hierarchical Out-of-Core Traversal of Rectilinear Grids,” *Proc. NSF/DoE Lake Tahoe Workshop Hierarchical Approximation and Geometrical Methods for Scientific Visualization,* Oct. 2000. Also LLNL Technical Report UCRL-JC-140581.
[26] V. Pascucci and R.J. Frank, “Global Static Indexing for Real-Time Exploration of Very Large Regular Grids,” *Proc. Supercomputing 2001,* Nov. 2001. Also LLNL Technical Report UCRL-JC-144754.
[27] M. de Berg and K.T.G. Dobrint, “On Levels of Detail in Terrains,” *Proc. ACM Symp. Computational Geometry,* pp. C26-C27, June 1995.
[28] M. Garland and P. Heckbert, “Fast Polygonal Approximation of Terrains and Height Fields,” Technical Report CMU-CS-95-181, Carnegie Mellon Univ., Sept. 1995.
[29] L. De Floriani, P. Magillo, and E. Puppo, “Efficient Implementation of Multi-Triangulations,” *Proc. IEEE Visualization ’98,* pp. 43-50, Oct. 1998.
[30] L. Velho and J. Gomes, “Variable Resolution 4-K Meshes: Concepts and Applications,” *Computer Graphics Forum,* vol. 19, pp. 195-212, Dec. 2000.
[31] T. Gerstner, M. Rumpf, and U. Weikard, “Error Indicators for Multilevel Visualization and Computing on Nested Grids,” *Computers & Graphics,* vol. 24, pp. 363-373, June 2000.
[32] F. Evans, S.S. Skiena, and A. Varshney, “Optimizing Triangle Strips for Fast Rendering,” *Proc. IEEE Visualization ’96,* pp. 319-326, Oct. 1996.
[33] S. Rusinkiewicz and M. Levoy, “QSplat: A Multiresolution Point Rendering System for Large Meshes,” *Proc. SIGGRAPH 2000,* pp. 343-352, July 2000.
[34] L.R. Willis, M.T. Jones, and J. Zhao, “A Method for Continuous Adaptive Terrain,” *Proc. 1996 Image Conf.,* pp. 23-28, June 1996.
[35] P. Lindstrom and V. Pascucci, “Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization,” Technical Report UCRL-JC-147847, Lawrence Livermore Nat’l Laboratory, May 2002.

Peter Lindstrom received the BS degree in computer science, mathematics, and physics from Elon College in 1994 and the PhD degree in computer science from the Georgia Institute of Technology in 2000. He is currently a computer scientist at the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research interests include geometric simplification and compression, multiresolution modeling, scientific visualization, and interactive rendering. He is a member of the IEEE.

Valerio Pascucci received the Laurea degree in ingegneria elettronica from the University “La Sapienza,” Rome, Italy, in 1993 and the PhD degree in computer science from Purdue University in 2000. He joined the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory in May 2000, where he is currently a research scientist and project leader. His research interests include progressive out-of-core techniques, analysis of scientific data sets, multiresolution geometric modeling, solid modeling, and computational geometry. He is a member of the IEEE.