Visualization of Large Terrains Made Easy

Peter Lindstrom Valerio Pascucci

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

Abstract

We present an elegant and simple to implement framework for performing out-of-core visualization and view-dependent refinement of large terrain surfaces. Contrary to the recent trend of increasingly elaborate algorithms for large-scale terrain visualization, our algorithms and data structures have been designed with the primary goal of simplicity and efficiency of implementation. Our approach to managing large terrain data also departs from more conventional strategies based on data tiling. Rather than emphasizing how to segment and efficiently bring data in and out of memory, we focus on the manner in which the data is laid out to achieve good memory coherency for data accesses made in a top-down (coarse-to-fine) refinement of the terrain. We present and compare the results of using several different data indexing schemes, and propose a simple to compute index that yields substantial improvements in locality and speed over more commonly used data layouts.

Our second contribution is a new and simple, yet easy to generalize method for view-dependent refinement. Similar to several published methods in this area, we use longest edge bisection in a top-down traversal of the mesh hierarchy to produce a continuous surface with subdivision connectivity. In tandem with the refinement, we perform view frustum culling and triangle stripping. These three components are done together in a single pass over the mesh. We show how this framework supports virtually any error metric, while still being highly memory and compute efficient.

1 INTRODUCTION

With a vast number of publications over the last several decades on level of detail creation and management of terrains and other height fields, the trend of increasing interactivity, visual quality, and I/O performance of published methods has generally come at the expense of algorithmic and implementation simplicity. Implementing a fully functional, integrated out-of-core visualization system, with support for accurate and fast view-dependent refinement, is in many cases a considerable time investment. To address this problem, we describe a simple to implement algorithm for improving the memory locality and minimizing the amount of data paging necessary, as well as a general framework for fast view-dependent refinement.

Our approach to handling large terrains is to lay out the data in an order that closely follows the order in which accesses to the vertices of the terrain are made. In essence, our goal is to find a fixed permutation of the list of all height field vertices that for a typical access pattern yields (near) optimal cache behavior. Using an indexing scheme that groups the mesh data by refinement level, we demonstrate a considerable improvement in paging performance. The issue of data paging to and from main memory is orthogonal to our approach. By virtue of greatly improved coherency, an explicit paging scheme is often not necessary. Rather, this task can be delegated to the operating system. Indeed, we make use of memory mapped files, using the `mmap` system call, to associate main memory address space with the terrain data stored on disk. This general framework allows different indexing schemes to be quickly integrated, while still leaving open the possibility of more sophisticated paging techniques for explicit data pre-fetching and management. Note that we are not tied to using memory mapping, but propose this as a simple, yet effective method for handling large terrains.

To accommodate fast display rates, it is essential to perform on-the-fly simplification of the high resolution mesh. Similar to recent algorithms for view-dependent refinement [8, 17, 19], we use recursive edge bisection as a means of refining a coarse base mesh defined over a regular grid of elevation points. Our method performs a top-down traversal of the mesh. A problem common to most algorithms based on top-down refinement is how to ensure that a consistent mesh without cracks is built. We guarantee this property by enforcing a nesting of the terms used in our view-dependent error metric, thereby implicitly forcing parent vertices to be introduced before their descendants. The details of this algorithm will be presented in Section 3. We here summarize some of its features:

Independence of error metric. The framework allows virtually any error metric to be incorporated. The supplemental information stored with the terrain data is essentially orthogonal to the choice of error metric. As a consequence, implementing a different metric requires only adding a handful of lines of code.

Memory efficiency. Per-vertex data is limited to position (parametric coordinates are optional), a scalar error term, and a scalar term for encoding a bounding sphere.

Computational efficiency. The refinement algorithm makes a single pass over the terrain, and is inherently output sensitive. That is, the time needed to construct a decimated mesh is proportional to the number of triangles it contains. The algorithm is also fast in practice, as only a few simple operations are involved.

Efficient view culling and triangle stripping. Fast, hierarchical view culling is supported at no additional memory cost. The recursive refinement of the terrain visits vertices in triangle strip order, allowing the rendering to be performed in tandem with the refinement.

Implicit mesh continuity. Due to the top-down approach and the guarantees made on parent-child relations, the mesh is unconditionally a continuous surface, both within and outside the view volume. Thus, no dependencies need to be maintained and no costly crack fixing has to be performed.

Near optimality. For a given accuracy, the top-down method produces meshes with only slightly more triangles than the minimal number obtained using bottom-up simplification.

Asynchronous updates. Because a continuous mesh that covers the entire terrain is always produced, refinement and view culling can be decoupled from the rendering stage. Using this approach, a dedicated rendering thread receives asynchronous mesh updates from a (possibly slower) refinement thread.

Simplicity. The refinement algorithm is very simple to implement, and requires only a few dozen lines of C code.

Address: 7000 East Avenue, L-560; Livermore, CA 94551; {pl,pascucci}@llnl.gov.

This work was performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48.

Before describing our algorithms, we will cover related work in terrain visualization. We conclude the paper with experimental results and directions for future work.

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 terrains, 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 done in the area of terrain visualization and level of detail creation and management. We will here limit our discussion to the more recent work on fine-grained, view-dependent simplification and refinement of terrain surfaces.

Gross et al. [13] 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. [17] 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, a coarse-grained simplification and refinement of rectangular blocks is made, followed by a fine-grained per-vertex decimation within each block. Due to this blocking, 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 non-trivial, and enforcing dependencies between vertices can be costly.

Based on the work by Lindstrom et al., Hoppe extended his work on *progressive meshes* to allow view-dependent refinement of arbitrary meshes [14]. This technique was later specialized for terrain rendering [15]. The run-time 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 [14], are still considerable. In addition, fully implementing his algorithm is not an easy task.

Using the same space of meshes as in [17], Duchaineau et al. [8] 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 pre-empted 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 [3, 10, 19, 23]. Most of these authors recognize the inherent complexity of doing input sensitive bottom-up simplification, and use simple heuristics for output sensitive top-down refinement. In this paper, we present improvements over some of these methods in several categories, including accuracy, mesh complexity, memory usage, refinement speed, generality, and, most importantly, ease of implementation.

2.2 Out-of-Core Paging and Data Layout

External memory algorithms [26], 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 structures that do not fit in the main memory of a single computer. New algorithmic techniques and analysis tools have been developed to address this problem, e.g. for geometric algorithms [1, 11, 18] and scientific visualization [2, 4].

In most terrain visualization systems [5, 6, 8, 12, 15–17, 19, 21] the external memory component is essential for handling real terrain and GIS databases. Hoppe [15] 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. [7] address the issue of external memory handling of large textures for terrain visualization. Reddy et al. [21] implemented a custom VRML browser specialized for terrain visualization, where efficiency is gained by combined use of multiresolution tiling, data caching, and predictive pre-fetching. The out-of-core component of the large-scale terrain system presented by Pajarola [19] is based on a decomposition of the domain into square tiles, which are stored in a database that supports fast 2D range queries. Efficient rendering is also achieved by organizing the set of triangles into a single strip that follows the Sierpinski space filling curve. A similar technique is used in our refinement algorithm.

Whereas the prevailing strategy for terrain paging has been to split the terrain up into large 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.

3 VIEW-DEPENDENT REFINEMENT

The goal of view-dependent refinement is to build a mesh with a small number of triangles that for a given view is a good approximation of the original, dense mesh. This construction is done continuously on-the-fly, and whenever the viewpoint changes the mesh is updated to reflect the change. To measure how well the coarse mesh approximates the original, one typically computes the deviation between corresponding points on the two meshes in object-space, and projects these errors onto the screen. Depending on whether the mesh is *simplified* bottom-up or *refined* top-down, triangles are merged or split to ensure that the projected errors meet some threshold or the mesh meets a given triangle budget.

In this section, we present a framework for performing top-down, view-dependent refinement of the terrain surface. We show how a single procedure can be used to efficiently perform the refinement, cull the mesh against the view volume, and simultaneously build a single triangle strip for the entire mesh. We first describe our main approach to refinement, and follow with details of how to implement each of its components.

3.1 Top-Down Mesh Refinement

There are two important classes of meshes used for view-dependent refinement: general, unstructured meshes (sometimes called triangulated irregular networks, or TINs), and regular (or semi-regular) meshes with subdivision connectivity. Whereas TINs have the potential to represent a surface with fewer triangles for a given error tolerance, the simplicity of regular subdivision hierarchies make them more appropriate for our purpose.

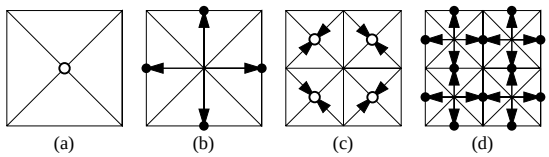


Figure 1: Edge bisection hierarchy. The arrows correspond to parent-child relationships in the directed acyclic graph of mesh vertices.

In our refinement algorithm, we use a particular type of subdivision based on *longest edge bisection* [8, 17, 23]. The meshes produced by this subdivision scheme, also called 4-k meshes [25], right-triangulated irregular networks [10], and restricted quadtree triangulations [19], have the property that they can be refined locally without having to maintain the entire mesh at the same resolution (see Figure 10, for example). In the edge bisection scheme, an isosceles right triangle is refined by bisecting its hypotenuse, thus creating two smaller right triangles (Figure 1). Starting with a coarse base mesh (typically two or four triangles), an adaptive, recursive refinement of the mesh is made. The refinement criterion, i.e. whether to split a triangle in two, is generally based on whether the triangle approximates the corresponding part of the original high resolution mesh well enough. For *view-dependent* refinement, this criterion also depends on factors such as the relative position and orientation of the viewer and the triangle.

The vertices introduced at the edge midpoints in the subdivision map directly to points on a regular, rectilinear grid. Thus it is natural to use the edge bisection hierarchy as a multiresolution representation for approximating height fields and terrain surfaces. 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 number of refinement levels.

It is also possible to perform the inverse of refinement—*simplification*—by starting with the highest resolution mesh and recursively merging pairs of triangles that satisfy a simplification criterion. A significant disadvantage of simplification versus refinement is that its computational complexity depends on the size of the highest resolution mesh, whereas the refinement complexity is linear in the size of the approximating mesh.

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 children j in the DAG corresponds to a triangle bisection, in which j is inserted on the hypotenuse and connected to i at the right-angle corner of the triangle (Figure 1). Thus, all non-leaf vertices not on the boundary of the mesh are connected to four children in the DAG, and have two parent vertices. 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 in the DAG. That is, for a vertex j to be active, its parents (and by induction all of its ancestors) must be active. Even when the DAG traversal is top-down, ensuring this property is not as easy as it may seem, since it is possible to reach j in the DAG without visiting one of its two parents. One solution to enforcing the validity of the mesh is to maintain explicit dependencies between each child and its parents; whenever a vertex is activated, the chain of dependencies is followed, and all ancestor vertices are activated [17]. However, this approach is inefficient, both in terms of computation and storage. Our approach, instead, is to satisfy Property 1 by ensuring that the error terms used in the refinement criterion are nested, thereby implicitly forcing all parent vertices to be activated with their descendants.

3.1.1 Refinement Criterion

The idea of using nested errors is not new. Blow [3] describes a method based on nested spheres. Each sphere is centered on the position $\mathbf{p}_i$ of a mesh vertex i, and represents the isocontour of i's projected screen space error $\rho_i = \rho(\delta_i, \mathbf{p}_i, \mathbf{e})$, where δ_i is an object (or world) space error term for i, and $\mathbf{e}$ is the viewpoint.[1] That is, ρ_i is constant for all viewpoints on the sphere's surface. 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 decimated quickly.

While theoretically simple, this method has a number of drawbacks. First, to ensure the nesting, τ must be fixed up-front. Second, the method is tied to a particular error metric; a metric based on distance alone. An orientation sensitive metric, such as the one in [17], does not necessarily lead to isosurfaces that have good nesting properties. Third, without maintaining explicit dependencies between vertices, or artificially inflating the spheres wherever necessary, Property 1 will generally not be satisfied, resulting in cracks in the mesh. Finally, every tree in the sphere forest must be visited during refinement. Since this forest can be arbitrarily large, further clustering of the trees may be necessary.

Our approach bears some resemblance to Blow's, but avoids many of its undesirable features. We, too, use a nested DAG of spheres, but for a different purpose, and its structure is given by the relationship between vertices in the refinement. In the discussion below, it is unimportant how the error terms δ and ρ are measured—we will discuss possible error metrics later in Section 3.1.2. However, we require that $\rho(\delta_i, \mathbf{p}_i, \mathbf{e})$ increases monotonically with δ_i when $\mathbf{p}_i$ and $\mathbf{e}$ are fixed (a reasonable assumption). Using these definitions, a sufficient condition for satisfying Property 1 is

$$\rho(\delta_i, \mathbf{p}_i, \mathbf{e}) \geq \rho(\delta_j, \mathbf{p}_j, \mathbf{e}) \qquad \forall j \in C_i$$

To guarantee this property, we could compute an adjusted 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, meaning that it would have to hold not only for i and its children, but also for all of its descendants. Visiting every descendant of each active vertex at run-time is clearly impractical for large terrains, since the set of descendants increases exponentially in size. Instead, we compute a lower bound $\rho_i^* \geq \rho_i$ by making use of our sphere hierarchy.

Note that ρ_i is made up of two distinct components: an object space error term δ_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

$$\delta_i^* = \begin{cases} \delta_i & \text{if } i \text{ is a leaf node} \\ \max\{\delta_i, \max\limits_{j \in C_i}\{\delta_j^*\}\} & \text{otherwise} \end{cases}$$

Then clearly $\delta_i^* \geq \delta_j^*$ for $j \in C_i$. Due to the monotonic relationship between ρ_i and δ_i, we must have $\rho(\delta_i^*, \mathbf{p}_i, \mathbf{e}) \geq \rho(\delta_i, \mathbf{p}_i, \mathbf{e})$, which ensures that there is no loss in visual accuracy. We don't necessarily have $\rho(\delta_i^*, \mathbf{p}_i, \mathbf{e}) \geq \rho(\delta_j^*, \mathbf{p}_j, \mathbf{e})$ for $j \in C_i$, however,

[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), and may depend on additional view information (e.g. gaze direction). It should be straightforward to generalize our definitions to such error metrics.

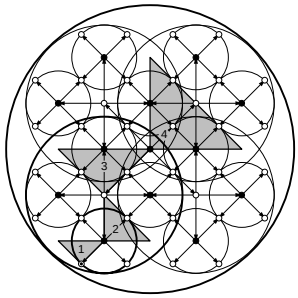


Figure 2: 2D analogue of the nested sphere hierarchy used for refinement and view culling. The four triangles are associated with the vertices at their right-angle corners. Notice that the bounding spheres do not completely contain their corresponding triangles on the bottom two levels in the DAG, but do contain them on level 3 and above.

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 $\mathbf{p}_i$). Essentially, we would like to add to $\mathbf{p}_i$ in the projection the set of $\mathbf{p}_j$ for all descendants j of i, and compute the maximum projection of δ_i^* from this set of points. Again, this is impractical, and we resort to a superset of points that is more easily expressed, defined by a ball B_i of radius r_i centered on $\mathbf{p}_i$. Define

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

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 Figure 2. Finally, we define the adjusted projected error as

$$\rho_i^* = \max_{\mathbf{x} \in B_i} \rho(\delta_i^*, \mathbf{x}, \mathbf{e})$$

Because $\delta_i^* \geq \delta_j^*$ and $B_i \supseteq B_j$, 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 run-time, we need to perform a constrained optimization over the ball B_i, which typically reduces to an optimization over the boundary of B_i. This may seem like an expensive process. However, it is generally easy to find a closed form expression for the minimum that can be pre-computed, and we will see in Section 3.1.2 how a metric based on distance alone can be expressed very concisely. It is interesting to note that this approach to computing lower error bounds is similar to the strategy used by Lindstrom et al. [17], in which an optimization over nested bounding boxes is done for coarse-grained simplification and refinement of large blocks of vertices.

During pre-processing of the data set, we compute δ^* and r for each vertex. In addition to the vertex's elevation (and coordinates in the parameter plane, if so desired), these are the only parameters needed in our top-down refinement algorithm. Given this general framework for refinement, we will now briefly discuss how to compute actual screen space errors for different error metrics.

3.1.2 Error Metrics

In this section, we consider possible object space (δ) and screen space (ρ) 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 [17, 23], but may be computed over triangles or even larger regions of influence associated with a vertex [8, 15]. Our framework accommodates both of these approaches, since the position or areal extent of the object space error can always be included in a vertex's bounding sphere by inflating it wherever necessary.

Object space errors can also be measured in relative terms, between two consecutive levels of refinement [17], or in absolute terms [8], with respect to the highest resolution mesh. The choice between relative and absolute errors is orthogonal to our refinement method, but should be made up-front since the errors need to be computed and propagated consistently during pre-processing.

Given an object space measure of error δ, a view-dependent algorithm projects δ onto the screen, resulting in a screen space error $\rho(\delta)$. While perspective projection is most commonly used in terrain visualization, it involves problems with singularities and can be somewhat computationally inefficient. Therefore it is common in view-dependent algorithms to substitute the distance along the view direction with the absolute distance, which is then used to attenuate the errors [8, 15, 17]. The most simple metric of this form can be written as

$$\rho_i = \lambda \frac{\delta_i}{\|\mathbf{p}_i - \mathbf{e}\|} \tag{2}$$

i.e. the projected error decreases with distance from the viewpoint. For the usual perspective projection onto a plane, $\lambda = \frac{w}{2\tan\varphi/2}$, where w is the number of pixels along the field of view φ. Equation 2 is in actuality a projection onto a sphere and not a plane, so a more appropriate choice for λ 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 ρ_i^* over a set of points $\mathbf{x} \in B_i$ (Section 3.1.1). For Equation 2 the maximum projection occurs where $\|\mathbf{x} - \mathbf{e}\|$ is minimized. For viewpoints inside B_i, this term is zero, and we activate i. If $\mathbf{e} \notin B_i$, then the minimum is $\|\mathbf{p}_i - \mathbf{e}\| - r_i$, and our maximum screen space error becomes

$$\rho_i^* = \lambda \frac{\delta_i}{\|\mathbf{p}_i - \mathbf{e}\| - r_i}$$

Comparing ρ_i^* against τ and rearranging and squaring some terms, we obtain

$$\begin{aligned} active(i) &\iff \lambda \frac{\delta_i}{\|\mathbf{p}_i - \mathbf{e}\| - r_i} > \tau \\ &\iff \frac{\lambda}{\tau}\delta_i > \|\mathbf{p}_i - \mathbf{e}\| - r_i \\ &\iff (\tfrac{1}{\kappa}\delta_i + r_i)^2 > \|\mathbf{p}_i - \mathbf{e}\|^2 \end{aligned} \tag{3}$$

where $\kappa = \frac{\tau}{\lambda}$ is constant during each refinement. The above expression involves only six additions and five multiplications, and is therefore very efficient to evaluate. Note that Equation 3 can be used whether the viewpoint is contained in i's bounding sphere or not. We use this as the default error metric in our terrain visualization system, and report the results of using it in Section 5.

If object space errors are measured vertically, then errors viewed from above appear relatively smaller than errors viewed from the side. Lindstrom et al. [17] describe an orientation sensitive metric that exploits this fact. While it is possible to derive a simple expression for such a metric for use with our refinement algorithm, empirical results observed by us and Hoppe [15] indicate that the reduction in mesh complexity over a purely distance based metric is on the order of a few percent. Therefore, we will not discuss this metric here in more detail.

3.1.3 Run-Time Refinement

With all the necessary pieces in hand, we now summarize the algorithm for top-down refinement and on-the-fly triangle strip construction. Pseudo-code for these steps is listed in Table 1. The refinement procedure builds a triangle strip $V = (v_0, v_1, v_2, \ldots, v_n)$, that is represented as sequence of vertex indices.[2] This triangle strip

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

```
tstrip-append(V, v, p)
1 if v ≠ v_{n-1} and v ≠ v_n then
2    if p ≠ parity(V) then
3       parity(V) ← p
4    else
5       V ← (V, v_{n-1})
6    V ← (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 mod 2)
4    submesh-refine(V, j, c_r(i, j), l − 1)

mesh-refine(V, n)
1 parity(V) ← 0
2 V ← (i_sw, i_sw)
3 submesh-refine(V, i_c, i_s, n)
4 tstrip-append(V, i_se, 1)
5 submesh-refine(V, i_c, i_e, n)
6 tstrip-append(V, i_ne, 1)
7 submesh-refine(V, i_c, i_n, n)
8 tstrip-append(V, i_nw, 1)
9 submesh-refine(V, i_c, i_w, n)
10 V ← (V, i_sw)
```

Table 1: Pseudo-code for recursive mesh refinement and triangle stripping.

construction is the same as in [17]. A vertex v is appended to the strip using the procedure tstrip-append (Table 1). Line 5 is used to "turn 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 [9]. The procedure submesh-refine corresponds to the innermost recursive traversal of the mesh hierarchy, where c_l and c_r are the two child vertices of the DAG parent j (p_g in Figure 5(b)) within the domain of the current triangle. We will discuss how to compute c_l and c_r in Section 4. Notice that submesh-refine 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. Because this evaluation constitutes a significant fraction of the overall refinement time, it is more efficient to move it one level up in the recursion, thereby evaluating it only once and then conditionally making the recursive calls.

Finally, the outermost procedure mesh-refine starts with a base mesh of four triangles (Figure 1(a)), 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, and $\{i_n, i_e, i_s, i_w\}$ the vertices introduced in the first refinement step (Figure 1(b)). The triangle strip is initialized with two copies of the same vertex to allow the condition on 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 interactive visualization and 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 [16, 22]. In this model, the render thread is periodically and asynchronously supplied with a list of geometry to render by the refinement thread. This "display list" is then used, and potentially reused over several frames, until a newly refined mesh is obtained. Our terrain visualization system allows this multi-threaded mode of rendering, in addition to the traditional sequential mode of processing.

3.2 View 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 simultaneously with the 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 [8], but is somewhat more efficient. In particular, we exploit the nested bounding sphere hierarchy to perform view culling, similar to [24]. 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 in theory for a small 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 is visible. By excluding i, a coarser triangle than t will be rendered. Note, however, that this can only happen at the periphery of the screen, and the resulting error of using a triangle of coarser resolution than t must be small in relation to the size of t. In practice, the bounding sphere hierarchy is intrinsically loose enough that these errors never occur above the second finest refinement level (see Figure 2), and we have seen no visible artifacts of culling the mesh.

The culling algorithm makes use of the six planes of the view frustum. The implicit plane equations for these are computed in object space coordinates and are passed along in the refinement. As in [8], we maintain one flag for each plane, 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 are necessary. If 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.

Figure 11 illustrates the advantage of performing view culling. From this figure, it is also evident that the mesh resolution drops rather sharply immediately outside the view volume. Still, some features towards the left edge of the mesh in Figure 11(b) remain, as they are too close to the top plane of the view frustum.

Note that because the bounding spheres are nested, the culling condition is consistent among parents and children, i.e. a child is visible only if its parents are. As a consequence, view culling does not introduce any T-junctions or cracks in the mesh—it always remains a continuous surface everywhere. This is a desirable feature when the refinement and render stages are asynchronous in that, regardless how much the refinement thread falls behind, the render thread always has a continuous mesh to display.

4 DATA LAYOUT AND INDEXING

This section addresses 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 to have a very simple mechanism to perform out-of-core paging of the data, while maintaining high performance. In particular, we take advantage of the paging mechanism of the operating system by using the `mmap` function.[3] `mmap` associates a part of the logical address space of the computer with a specific disk file. Using this mechanism the external memory part of our implementation consists simply of a call to `mmap` to associate the memory address of an array with the terrain information (elevation values, precomputed errors, etc.) stored on disk. After this step the array of terrain vertices 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 one 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 pre-fetching mechanisms that might further improve the out-of-core performance of the terrain traversal.

Given the framework described above, the external memory processing problem can be reduced to a data layout problem. We know the structure of the terrain traversal algorithm, and we have a mechanism that hides the need for data paging from the application. Using this framework, we need to determine: (i) a way of storing the raw data that minimizes paging events, and (ii) an efficient procedure for computing the index of the data element in the given refinement order, so that no significant added cost is introduced in the refinement process.

The following two subsections describe a data layout scheme that satisfies requirements (i) and (ii), and that has a particularly straightforward implementation.

[3] The equivalent Windows function is called `MapViewOfFile`.

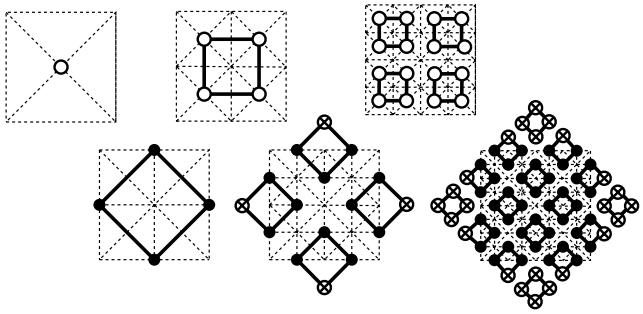


Figure 3: Top row: First three levels of the white quadtree. Bottom row: A complete black quadtree is obtained by adding the crossed ghost vertices to it.

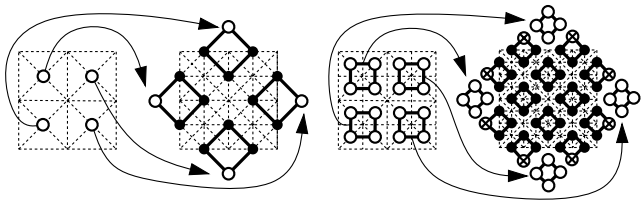


Figure 4: Illustration of embedding the top two levels of the white quadtree in the unused parts of the black quadtree.

4.1 Interleaved Quadtrees

On the basis of the edge bisection refinement algorithm, each vertex (apart from the four corners of the grid) can be labeled as white, if introduced at an odd level of refinement, or black, if introduced at an even level. Figure 1 shows this classification for the first four levels of refinement. The top row of Figure 3 shows how the sequence of white vertices forms a quadtree—the *white quadtree*, Q_w. Each white node is in fact the center of a square tile in a quadtree decomposition of the rectilinear grid. Interestingly the black vertices can also be considered as part of a *black quadtree*, Q_b. Figure 3 shows as crossed circles the vertices that need to be added outside the rectilinear grid to form a complete black quadtree. We will refer to these additional vertices as "ghost vertices." The black quadtree is rotated 45 degrees with respect to the white quadtree. Note that Q_b does not start at the root but at the first level of refinement. Adding a virtual root node makes Q_b one level taller than Q_w.

Since the traversal of the DAG (see Section 3.1) is performed top-down, starting form the root, good data 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; vertices that are geometrically close should be stored close together in memory. For a quadtree, this can be achieved by using the order induced by the following formula that computes the index $c(p, k)$ of the kth child of the parent node p:

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

where m is a constant dependent on the index of the root and the index distance between consecutive levels of resolution. Using this data layout, all the vertices on the same level of resolution are stored together, starting with the coarsest level. The index distance between two vertices on the same 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 the white quadtree, with roots $r_b = 3$ and $r_w = 4$. Since r_b is not used in practice, we can assign the first four indices (from 0 to 3) to the corners of the grid. The first child of r_b is stored immediately after r_w, and we have $c(r_b, 0) = 4 \cdot 3 + 0 + m = 5$ and $c(r_w, 0) = 4 \cdot 4 + 0 + m = 9$, which both imply $m = -7$. Notice in Figure 3 that the ghost vertices in Q_b are not used. Because the data is eventually stored as a single linear array, this results in unwanted "holes" in the array.

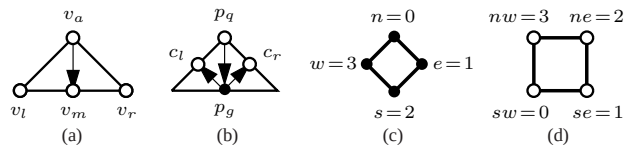


Figure 5: Naming conventions for the nodes in the refinement hierarchy. (a) Refinement of a single triangle using linear indexing. (b) Refinement using the interleaved quadtrees. (c, d) Sibling vertices in the black and white quadtrees, respectively.

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. Figure 4 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 $m = -11$.

One drawback of our quadtree-based indexing schemes is that they use a non-contiguous address space. In the case of interleaved quadtrees, the unused ghost vertices result in a waste in storage resources of roughly 66% of the input data. This overhead is reduced to 33% in the storage layout where the white quadtree is embedded in the black quadtree. This overhead can be completely eliminated by using a data layout based on a hierarchical version of the Lebesgue Z-order space filling curve. Because the implementation of this scheme is not as straightforward as the quadtree-based schemes described above, we do not provide the details of the computations involved in this indexing scheme, but refer the interested reader to [20]. In Section 5 we include empirical results of the performance achieved both with the quadtree-based schemes discussed here and with the hierarchical Z-order space filling curve.

4.2 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 (Figure 5(a)). 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 edge (v_l, v_r), which can be computed simply as the average $v_m = \frac{1}{2}(v_l + v_r)$.

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 Figure 5(b). The new white vertices c_l (left child) and c_r (right child) have a common black *graph parent* p_g in the refinement DAG. Moreover the graph parent of p_g is also the *quadtree parent* p_q of c_l and c_r. Based on this observation, the indices c_l and c_r can be computed from the index of their quadtree parent p_q using Equation 4. The relative positions of p_q and p_g determine which two branches (the values of the index k) need to be used to reach c_l and c_r from p_q. In numbering the four children of a common parent, we use the conventions shown in Figure 5(c, d). We have chosen these particular labels for the branches carefully to allow an efficient child index computation without having to use any lookup tables. Using these conventions, we can compile the two transition tables shown in Table 2. Note that the value of k can be determined from the lowest two bits of the vertex index. Because of considerable redundancy in the transition tables, and due to our choice of branch labels, the transition tables, and consequently the child indices c_l and c_r, can

k_l, k_r		p_g			
		n	e	s	w
p_q	sw	se, ne	ne, nw	nw, sw	sw, se
	se	nw, sw	sw, se	se, ne	ne, nw
	ne	se, ne	ne, nw	nw, sw	sw, se
	nw	nw, sw	sw, se	se, ne	ne, nw

k_l, k_r		p_g			
		sw	se	ne	nw
p_q	n	e, s	s, w	w, n	n, e
	e	w, n	n, e	e, s	s, w
	s	e, s	s, w	w, n	n, e
	w	w, n	n, e	e, s	s, w

Table 2: Transition tables used to determine the left (k_l) and right (k_r) branch to be used in Equation 4 for quadtree parent p_q and DAG parent p_g. The symbols in the tables and their values follow the conventions of Figure 5(c, d).

be expressed concisely using a few arithmetic operations:

$$c_l(p_q, p_g) = 4p_q + ((2p_q + p_g + m + 1) \bmod 4) + m$$
$$c_r(p_q, p_g) = 4p_q + ((2p_q + p_g + m + 2) \bmod 4) + m$$

These simple equations are used in the submesh-refine procedure in Section 3.1.3.

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, which is made up of $16,385 \times 16,385$ vertices at 10 meter horizontal and 0.1 meter vertical resolution.[4] This data set occupies roughly 5 GB on disk. 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 3.1.2 for all results presented here. To evaluate the efficiency in mesh complexity for a given accuracy, we recorded for 1,000 views the number of rendered triangles obtained using both a bottom-up simplification of the terrain (which produces the minimal number of triangles for a given threshold) and our top-down scheme. Figure 6 shows a histogram of the distribution of views in which our suboptimal mesh contained a certain percentage of triangles beyond the minimum possible. Our mesh generally contains more triangles than necessary due to the requirement of nested errors and occasional inflated error terms. It is interesting to note, however, that the top-down method produces meshes that are just a few percent larger than minimal for a one-pixel error tolerance.

We next evaluate the performance increase due to the use of culling and multi-threading (one thread each for rendering and refinement). These results are summarized in Table 3 and plotted in Figure 13. The graph for multi-threading with culling in Figure 13(a) corresponds to the first sequence shown in the accompanying video. We were able to maintain 60 frames per second during nearly the entire fly-over. When the number of rendered triangles exceeded 50,000, however, the frame rate slowed briefly. Based on these numbers, we expect our algorithm to surpass the performance of Hoppe's method [15]. He reports a rendering speed of 8,000 triangles at 60 Hz on an SGI Onyx InfiniteReality, whereas using multi-threading we were able to sustain 40,000 rendered triangles

[4]We obtained a subset of the freely available data from http://duff.geology.washington.edu/data/raster/tenmeter/bil10/.

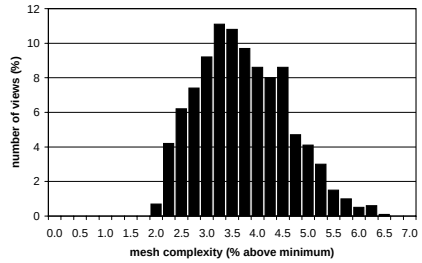


Figure 6: Mesh complexity distribution for the top-down scheme in relation to the optimal mesh over 1,000 different views. The error tolerance is $\tau = 1$ pixel.

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

Table 3: Flight time and average performance for 2816-frame fly-over (see also Figure 13). The rendering performance is measured as the number of rendered triangles over the frame time, which includes the refinement time in single-threaded mode.

at the same rate. Figure 13(a) also demonstrates a clear advantage of using both culling and multi-threading.

Figure 13(b) highlights the refinement performance, with and without culling, measured in 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 towards 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.

Finally, we evaluated the efficiency of using a single triangle strip. We found that the ratio of triangle strip vertices to the number of non-degenerate triangles averaged 1.56 vertices/triangle, with virtually no variance. This number should be compared to 3 vertices/triangle for a list of independent triangles.

5.2 Data Layout

In this section, we compare the memory performance of four different indexing schemes: the single quadtree scheme from Section 4.1, where the "white" tree is embedded in the "black" tree; the Z-order indexing scheme; a blocking scheme based on 32×32 tiles from the highest resolution data; and a standard matrix layout in row major form. For all these methods, we stored the fields $(\mathbf{p}, \delta^*, r)$, which together occupy 20 bytes, with each vertex (see Section 3). Our focus here is not on the storage efficiency of the vertex records—it is entirely possible to compress or even eliminate some fields in this record. Rather, we assume fixed-length records and focus on how efficient the different indexing schemes are at accessing them.

Figure 7 shows the total number of page faults, after executing the same flight path as in the video, for varying values of the error tolerance τ. Smaller values of τ result in larger meshes being rendered and more data being paged in. Clearly, the hierarchical indexing schemes (quadtree-based and Z-order) greatly outperformed the linear and block-based schemes, and often lead to drastically improved paging speeds (Figure 12). Perhaps surprising, 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

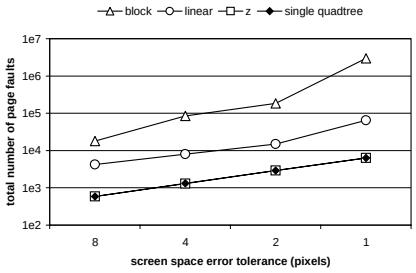


Figure 7: Total number of page faults vs. screen space error tolerance τ on 900 MB SGI.

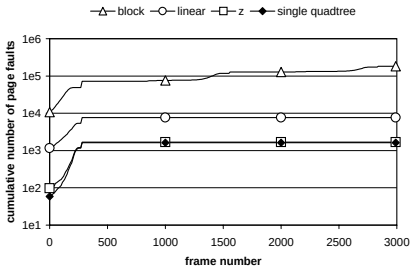


Figure 8: Cumulative number of page faults over time on 900 MB SGI.

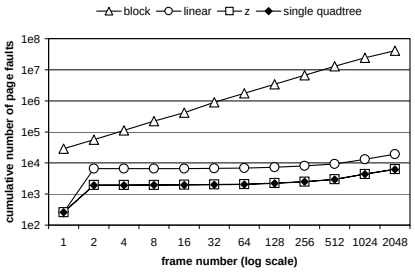


Figure 9: Cumulative number of page faults over time on 64 MB PC.

groups of many 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 subsample the data and create a multiresolution pyramid, allowing more coherent access to different resolutions of data. However, such an indexing scheme uses multiple indices for each vertex, which would arguably make for an unfair comparison with our other indexing schemes.

We also investigated the paging behavior over time. Results for the SGI Octane are shown in Figure 8, while the PC results are shown (on a log-log scale) in Figure 9. These graphs show that there is a significant hit at startup, when no data is memory resident, after which pages stay in user memory or can be reclaimed quickly from the operating system's cache. Surprisingly, the quadtree-based scheme, which requires 33% of unused space, performs slightly better than the Z-order scheme (Figure 8). Unfortunately we have no plausible explanation for this behavior.

Finally, we measured the raw in-core refinement speed of all indexing schemes. Due to better cache locality, the 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 Z-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 AND FUTURE WORK

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. Our emphasis has been on the simplicity and efficiency of these methods—the core of these algorithms can be implemented in as little as a few dozen lines of C code. In spite of their simplicity, the rendering and paging speed of our algorithms compete with the state of the art in terrain visualization.

We see several avenues for future work. As demonstrated in [25], there is no need for the refinement to insert vertices at edge midpoints. Using our current data structures, which store the xyz-coordinates with each vertex, we could perform a data-dependent triangulation that still has the same subdivision connectivity. This is particularly desirable for representing features such as roads and rivers. So far, we have not employed *geomorphing* to smooth out transitions in mesh resolution, even though we have not found these transitions to be particularly distracting. We believe that our framework could easily support geomorphing with no significant code changes. We would also like to compare and integrate different error metrics and indexing schemes into our framework. While contrary to the spirit of our approach, more work needs to be done to address the issue of paging efficiency. By using prediction and prefetching, it may be possible to further improve the rate at which data is paged in and integrated with the multiresolution terrain.

References

[1] L. Arge and P. B. Miltersen. On Showing Lower Bounds for External-Memory Computational Geometry Problems. *External Memory Algorithms and Visualization*. American Mathematical Society Press, 1999.

[2] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel Accelerated Isocontouring for Out-of-Core Visualization. *Proceedings of the 1999 IEEE Parallel Visualization and Graphics Symposium*, 97–104. Oct. 1999.

[3] J. Blow. Terrain Rendering at High Levels of Detail. *Proceedings of the 2000 Game Developers Conference*. Mar. 2000.

[4] Y.-J. Chiang and C. T. Silva. I/O Optimal Isosurface Extraction. *IEEE Visualization '97*, 293–300. Nov. 1997.

[5] D. Clark and M. Bailey. Visualization of Height Field Data with Physical Models and Texture Photomapping. *IEEE Visualization '97*, 89–94. Nov. 1997.

[6] D. Davis, T. Y. Jiang, W. Ribarsky, and N. Faust. Intent, Perception, and Out-of-Core Visualization Applied to Terrain. *IEEE Visualization '98*, 455–458. Oct. 1998.

[7] J. Döllner, K. Baumann, and K. Hinrichs. Texturing Techniques for Terrain Visualization. *IEEE Visualization 2000*, 207–234. Oct. 2000.

[8] 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. *IEEE Visualization '97*, 81–88. Nov. 1997.

[9] F. Evans, S. S. Skiena, and A. Varshney. Optimizing Triangle Strips for Fast Rendering. *IEEE Visualization '96*, 319–326. Oct. 1996.

[10] W. Evans, D. Kirkpatrick, and G. Townsend. Right-Triangulated Irregular Networks. *Algorithmica*, 30(2):264–286, Mar. 2001.

[11] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-Memory Computational Geometry. *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*. Nov. 1993.

[12] C. Gotsman and B. Rabinovitch. Visualization of Large Terrains in Resource-Limited Computing Environments. *IEEE Visualization '97*, 95–102. Nov. 1997.

[13] M. Gross, R. Gatti, and O. Staadt. Fast Multiresolution Surface Meshing. *IEEE Visualization '95*, 135–142. Oct. 1995.

[14] H. Hoppe. View-Dependent Refinement of Progressive Meshes. *Proceedings of SIGGRAPH 97*, 189–198. Aug. 1997.

[15] H. Hoppe. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. *IEEE Visualization '98*, 35–42. Oct. 1998.

[16] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, A. O. den Bosch, and N. Faust. An Integrated Global GIS and Visual Simulation System. *Tech. Rep. GIT–GVU–97–07*, Georgia Institute of Technology, Mar. 1997.

[17] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. Turner. Real-Time, Continuous Level of Detail Rendering of Height Fields. *Proceedings of SIGGRAPH 96*, 109–118. Aug. 1996.

[18] Y. Matias, E. Segal, and J. S. Vitter. Efficient Bundle Sorting. *Proceedings of the 11th Annual SIAM/ACM Symposium on Discrete Algorithms*, 839–848. Jan. 2000.

[19] R. B. Pajarola. Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation. *IEEE Visualization '98*, 19–26. Oct. 1998.

[20] V. Pascucci and R. J. Frank. Global Static Indexing for Real-time Exploration of Very Large Regular Grids. *Proceedings of Supercomputing 2001*. Nov. 2001. To appear. Available as LLNL technical report UCRL-JC-144754.

[21] M. Reddy, Y. Leclerc, L. Iverson, and N. Bletter. TerraVision II: Visualizing Massive Terrain Databases in VRML. *IEEE Computer Graphics & Applications*, 19(2):30–38, Mar. - Apr. 1999.

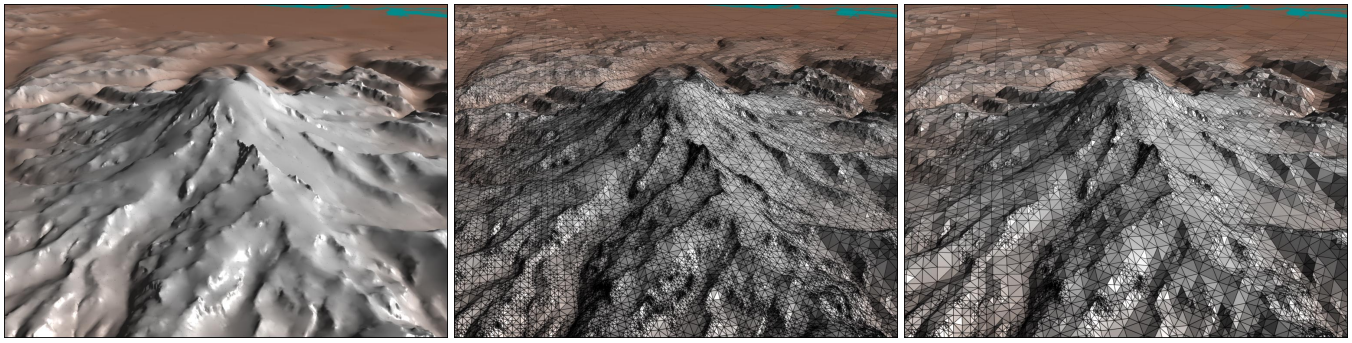
[22] J. Rohlf and J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Proceedings of SIGGRAPH 94*, 381–395. Jul. 1994.

[23] S. Röttger, W. Heidrich, P. Slussallek, and H.-P. Seidel. Real-Time Generation of Continuous Levels of Detail for Height Fields. *Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization*, 315–322. Feb. 1998.

[24] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. *Proceedings of SIGGRAPH 2000*, 343–352. Jul. 2000.

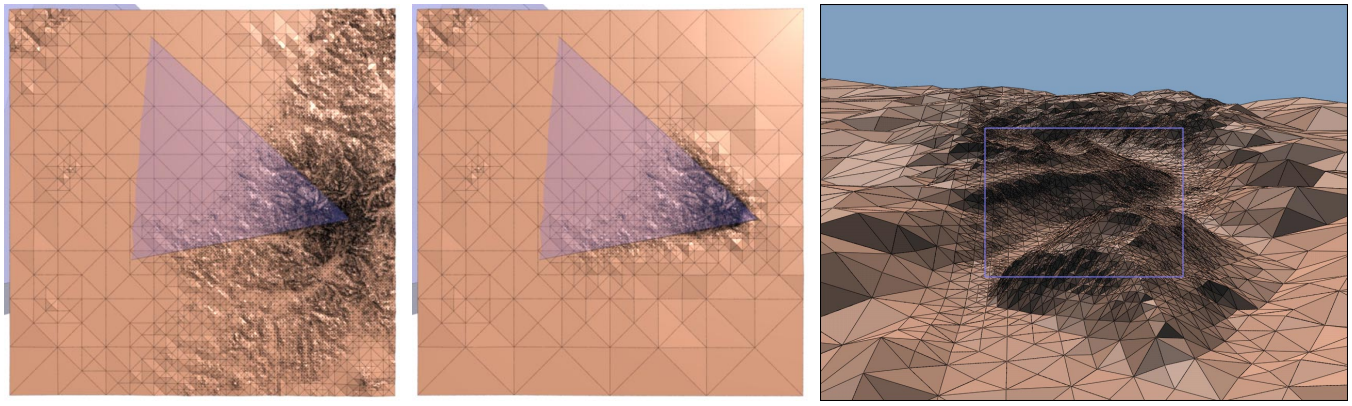
[25] L. Velho and J. Gomes. Variable Resolution 4-k Meshes: Concepts and Applications. *Computer Graphics Forum*, 19(4):195–212, Dec. 2000.

[26] J. S. Vitter. External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 2001. To appear.



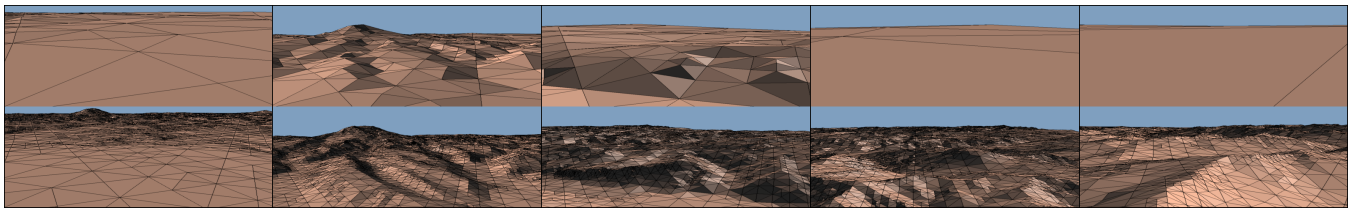
(a) $\tau = 2$ pixels; 79,382 triangles. (b) $\tau = 2$ pixels; 79,382 triangles. (c) $\tau = 4$ pixels; 25,100 triangles.

Figure 10: View of Mount Rainier, Washington. (b, c) Edge bisection subdivision meshes for two different screen space error thresholds τ.



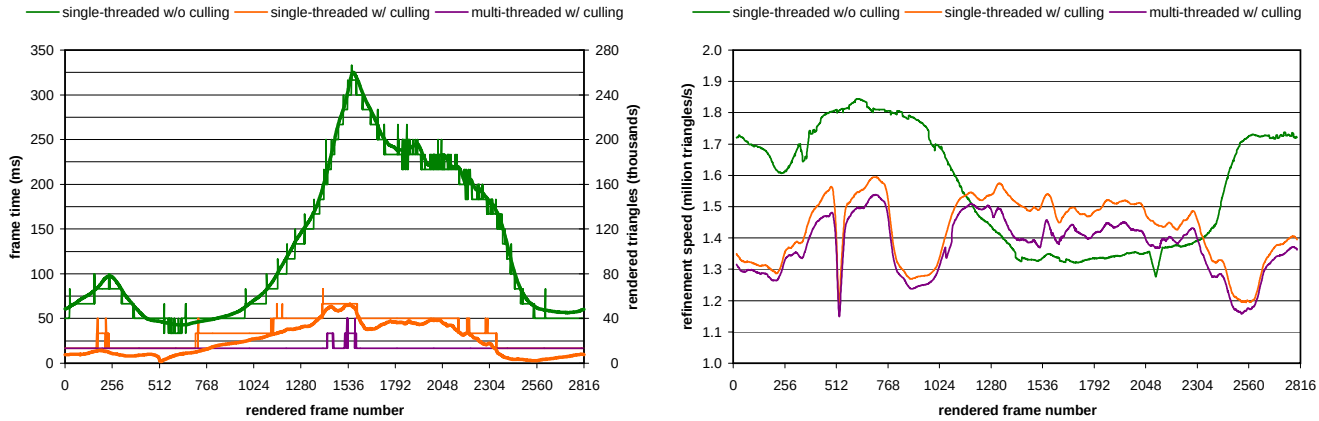
(a) Without view culling; 97,435 triangles. (b) With view culling; 25,100 triangles. (c) View culling against rectangle.

Figure 11: Examples of view frustum culling. The mesh is everywhere C^0 continuous, whether culled or not. (a, b) The view frustum is shown in semi-transparent violet, with the viewer looking across the terrain from the right. This view is the same as in Figure 10. (c) The mesh resolution drops quickly outside the view frustum (shown as a violet rectangle).



(a) Frame #1000. (b) Frame #1350. (c) Frame #1850. (d) Frame #2000. (e) Frame #2200.

Figure 12: Frames from two multi-threaded fly-over sequences using linear, row major indexing (top) and quadtree-based indexing (bottom). The flight paths for the two sequences are the same. The improved cache performance of the quadtree-based scheme results in more detail being paged in more quickly.



(a) Frame time (thin lines) and number of rendered triangles (thick lines). (b) Refinement performance over time.

Figure 13: In-core rendering and refinement performance on SGI Onyx2 over several thousand frames during a terrain fly-over (see accompanying video). The curves correspond to the use of single-threading, with and without culling, and multi-threading with culling. The hierarchical indexing scheme was used in all three runs. (a) Using multi-threading a steady 60 Hz is maintained during nearly the entire flyover. The number of triangles for the two schemes that used culling coincide, therefore the graph for only one of them is shown. (b) The vertical axis corresponds to the number of non-degenerate triangles in the triangle strip divided by the (wall clock) refinement/view culling time.