Adaptive 4-8 Texture Hierarchies

Lok M. Hwa* — University of California, Davis
Mark A. Duchaineau† — Lawrence Livermore National Laboratory
Kenneth I. Joy‡ — University of California, Davis

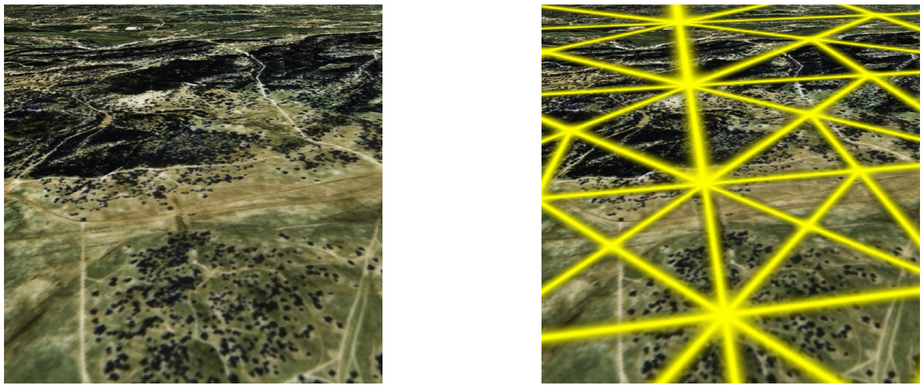

Figure 1: Two screen shots of an overflight of Fort Hunter Liggett, CA that illustrate the use of 4-8 texture hierarchies. On the left is the seamless textured image produced by the system, while the right shows the outline of the texture tiles used in producing the image.

Abstract

We address the texture level-of-detail problem for extremely large surfaces such as terrain during realtime, view-dependent rendering. A novel texture hierarchy is introduced based on 4-8 refinement of raster tiles, in which the texture grids in effect rotate 45 degrees for each level of refinement. This hierarchy provides twice as many levels of detail as conventional quadtree-style refinement schemes such as mipmaps, and thus provides per-pixel view-dependent filtering that is twice as close to the ideal cutoff frequency for an average pixel. Because of this more gradual change in low-pass filtering, and due to the more precise emulation of the ideal cutoff frequency, we find in practice that the transitions between texture levels of detail are not perceptible. This allows rendering systems to avoid the complexity and performance costs of per-pixel blending between texture levels of detail.

The 4-8 texturing scheme is integrated into a variant of the Realtime Optimally Adapting Meshes (ROAM) algorithm for view-dependent multiresolution mesh generation. Improvements to ROAM included here are: the diamond data structure as a streamlined replacement for the triangle bintree elements, the use of low-pass-filtered geometry patches in place of individual triangles, integration of 4-8 textures, and a simple out-of-core data access mechanism for texture and geometry tiles.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing Algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric Algorithms, Object Hierarchies; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality

Keywords: Large Data Set Visualization, Level-of-Detail Techniques, View-Dependent Visualization, Adaptive Textures, Out-of-Core Algorithms

*e-mail: lmhwa@ucdavis.edu
†e-mail: duchaineau1@llnl.gov
‡e-mail: kijoy@ucdavis.edu

1 Introduction

Graphics hardware has become orders of magnitude faster and cheaper in recent years, yet there remains a strong need to render textured geometry from databases containing far more detail than can be displayed in realtime. A classic motivating example is terrain visualization, in which photo-imagery and elevation data are available on planetary scales, resolving to ten meters or better on average, with meter or sub-meter data available in some regions (such as the one-meter database of Fort Hunter Liggett, CA, shown in Figure 1). With new data collection instruments and data handling capabilities, this wealth of information is likely to grow rapidly. The NASA MOLA data, for example, covers Mars at a resolution of 128 elevation bins per degree, totaling around one billion elevations [1]. Publicly available data from the USGS covers the state of Washington at 10 meter horizontal and 10cm vertical spacing, totaling 1.4 billion elevation values [19]. Dynamic, view-dependent adaptations of geometric meshes and texture tile hierarchies are required to provide fast and accurate renderings of these large-scale terrain databases.

Since hardware rendering rates have grown to exceed 200 million triangles per second, this means that choosing triangle adaptations for uniform screen size will result in roughly one-pixel triangles for full-screen display at 100 frames-per-second rendering rates. At this point it is no longer desirable to make triangles non-uniform in screen space due to variations in surface roughness, since this will only lead to sub-pixel triangles and artifacts. This situation for geometry is now in a similar regime to that of texture level-of-detail adaptation, which seeks to make each texel project to roughly one pixel in screen space. Overall then our goal is to low-pass filter the geometry and textures so that triangles and texels project to about a pixel.

While many geometric hierarchies have been devised for large-data view-dependent adaptation, the above analysis suggests that uniform aspect-ratio triangles are more desirable for attaining better control of geometric antialiasing. Also, better low-pass filtering methods are known for regular grids. Texture hierarchies are more constrained than geometry, since graphics hardware works most effectively with raster tiles of modest, power-of-two sizes. For efficiency of texture loading and packing, we avoid consideration of texture atlas schemes in which a power-of-two tile is filled with irregular sub-regions that are used independently. This leads us to use regular grids for efficiency and uniformity of treatment. In theory, there are only two regular tilings of the plane that allow conformant adaptive meshes to be formed without special fix-ups at level of detail transitions: the 4-8 meshes and the 4-6-12 meshes [7, 8]. We chose the 4-8 meshes, shown in Figure 2, since these match the constraints of texture hardware and have many known desirable properties [12, 6, 13].

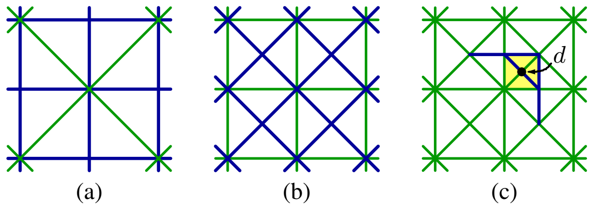

Figure 2: A 4-8 mesh illustrating different levels of resolution. Part (a) shows a course, uniform refinement, which is effectively a grid of squares (blue) with distinguished diagonals (green). Part (b) is one level finer everywhere. Note the blue squares are rotated $45°$ and scaled by $\sqrt{1/2}$. Part (c) shows the selective refinement of (b) to add the diamond (yellow) with center d.

While several data structures have been devised to support 4-8 refinement, we found that additional streamlining and unification was possible. This paper introduces a *diamond* data structure, in which each diamond element simultaneously has unique associations with a vertex (its center), an edge (its distinguished diagonal), and a quadrilateral face of a 4-8 refinement mesh. A diamond represents the pairing of two right isosceles triangles at the same level of detail in the 4-8 mesh that share a base edge. Since basic operations on the 4-8 mesh must treat these diamonds as a unit, it is logical and efficient to use the diamond as the backbone data structure rather than bintree triangles. Section 3 provides details on the diamond structure and its use in 4-8 incremental mesh adaptation.

Both geometry and textures are treated as small regular grids, called *tiles*, defined for each diamond in the hierarchy. Tiles at a level of resolution matching the input data are either copied or resampled. Coarser tiles are computed using low-pass filtering in an out-of-core traversal. Finer tiles can be obtained using 4-8 subdivision [23] with the optional addition of procedural detail. For efficient input and output, files and disk blocks are laid out using a diamond indexing scheme based on the Sierpinski space-filling curve. Tiles are described in Section 4. Sierpinski indexing, and out-of-core preprocessing are described in Section 5.

For geometric rendering, *patches* of 256 or 1024 triangles are stored as indexed vertex arrays in Sierpinski order for highly efficient rendering on graphics hardware. Using uniform refinement, any power of four increase in triangle count will result in conformant meshes [18, 11]. We are able to achieve triangle throughput close to the practical limits on recent PC video cards. Section 6 outlines how patches are laid out and updated.

The adaptive 4-8 textures, defined in detail in Section 7, fill each diamond area with a regular-grid image raster, rendered using bilinear interpolation. Neighboring tiles share boundary samples on their mutual edges, and the 4-8 mesh refinement naturally defines a parent-child grid-structure relationship suitable for various filtering operations. We allow each ROAM leaf triangle patch to independently choose which texture level-of-detail to map to, based on its estimated pixel area for the current view transform. A mapping from the triangle patches' parameterization to the texture diamond's parameter space is computed as needed when this level-of-detail selection changes. This change requires an update of the vertex array texture coordinate data stored in special graphics hardware memory (e.g. AGP memory), which is an expensive operation that can require synchronization with previously launched asynchronous rendering activity. Therefore the triangle-patch texture level-of-detail updates are budgeted per frame based on similar dual-queue operations used by the ROAM algorithm.

Overall this approach to forming tile hierarchies and accessing them during frame-to-frame incremental updates results in a visually seamless, high quality display of arbitrarily large terrain and imagery databases. Some implementation details and numerical results are presented in Section 8, but the ultimate proof is to see the system in action on a huge data set. The visual appearance is in our experience consistently very high. Indeed, we were pleasantly surprised that no per-pixel blending of texture level-of-detail seems to be needed; we believe this is due to the gradual factor-of-two changes in information content between levels.

2 Related Work

A great variety of geometric level-of-detail algorithms have been devised for realtime rendering of massive terrains and other data sets. An overview of many historical methods can be found in [16]. The most common means of organizing geometry are Triangulated Irregular Networks (TINs) [20, 9], and Hierarchies of Right Triangles (HRTs) [15, 7]. Generally the HRT methods can be implemented to have greater performance and lower memory use per triangle, but require a modest increase in triangle budget to achieve the same accuracy [7]. For the reasons outlined earlier, we focus on regular-grid representations and HRT view-dependent adaptations, and review the relevant papers here.

First it is important to note that HRTs are equivalent to adaptive 4-8 meshes. An early paper using HRTs for view-dependent dynamic meshing was Lindstrom *et al.* [12]. They utilize an elegant block-adaptive refinement using frame-to-frame coherence, followed by a fine-grained bottom-up vertex-reduction method to reduce the size of the mesh for display purposes. Duchaineau *et al.* [6] introduce a dual-queue algorithm (ROAM) to incrementally split and merge HRT elements while maximizing the use of frame-to-frame coherence for frustum culling, priority computations, mesh updates and triangle stripping. Lindstrom and Pascucci [13] simplify the overall HRT processing to a minimal triangle bintree recursion per frame that requires no special effort to maintain crack-free meshes, produces a single generalized triangle strip as output, and uses a novel vertex indexing scheme to automatically make out-of-core access efficient using an existing operating-system virtual memory system. They extend this [14] to allow smoother view-dependent meshes through interpolation, and test additional space-fill indexing strategies. Gerstner uses Sierpinski indexing for triangles, and identifies the resulting duplicate indices using a simple state machine. The method is intended for use during recursive traversal of the triangle bintrees, and requires explicit links in the vertex database to avoid gaps in the disk or memory layout. Pajarola [17] utilizes a restricted quadtree triangulation, similar to an adaptive 4-8 mesh, for terrain visualization. Pomeranz [18] demonstrates how the ROAM algorithm can be extended to utilize pre-computed HRT patches in place of individual triangles to better exploit modern graphics hardware while maintaining crack-free triangulations. Levenberg [11] extends this further by allowing HRT patches to be computed dynamically during interaction.

Large texture processing has been attempted by several researchers. Williams [24] introduces the *mipmap* method of pre-filtering texture levels of detail, which are images of increasingly reduced resolution arranged as a pyramid. Starting with the finest level, each coarser level represents the image using one quarter the number of texels (half the number of texels in each dimension). Per-pixel rendering with a mipmap is accomplished by projecting the pixels into mipmap space using texture coordinates and camera transformations. Typically a rendered pixel is colored using a variant of trilinear interpolation of eight texels taken from two adjacent levels of the mipmap hierarchy.

Tanner *et al.* [21] introduce clipmaps, an extension of mipmaps, that also utilizes a factor-of-four texture pyramid, but allows arbitrarily large out-of-core textures to be paged into the in-memory pyramid. This algorithm utilizes the fact that a complete mipmap pyramid is rarely used during the rendering of a single image (particularly in terrain rendering), and much of the pyramid can be clipped away, allowing much larger textures to be used.

Ulrich [22] combines a quadtree of mipmap and geometry tiles, called *chunks*, to handle out-of-core view-dependent meshing and texturing of huge terrains. The texture and geometry chunks are produced in a preprocessing step and are static during runtime interaction. Geometry chunks are based on adaptive 4-8 refinement, with special "flanges" to hide the tiny cracks that occur at chunk boundaries. Each chunk is stored in special graphics memory and can be rendered with a single draw call. The chunks are refined based on the viewpoint to meet the desired visual fidelity, and are paged from disk. Similarly, mipmap tiles are loaded and accessed from the geometry chunks based on calculations of maximum pixel size in the mipmap. This determines the finest level of detail that will be used in a mipmap, and by refining the mipmap tiles accordingly, the mipmap per-pixel blending will automatically generate seamless texture imagery across tile boundaries.

Further research by Döllner *et al.* integrates clipmap-like behavior with terrain rendering by using memory-mapped texture files [5]. Their method utilizes a multiresolution texture system that works in conjunction with a multiresolution model for the terrain geometry. They build a tree of texture patches that is closely associated with the hierarchical model of the terrain geometry. The rendering algorithm simultaneously traverses the multiresolution model for terrain geometry and texture trees, selecting geometry patches and texture patches according to a user-defined visual error threshold. However, their method utilizes in-core quadtrees for texture storage, resulting in a power-of-four texture hierarchy.

Cignoni *et al.* [2, 3, 4] have demonstrated the ability to display both adaptive geometry and texture of large terrain data sets in real-time. They utilize a quadtree texture hierarchy and a bintree of triangle patches (TINs) for the geometry. The triangle patches are constructed off-line with high-quality simplification and triangle stripping algorithms, and are selectively refined from scratch each frame. Textures are managed as square tiles, organized as a quadtree. The rendering system traverses the texture quadtree until acceptable error conditions are met, and then traverses the corresponding patches in the geometry bintree system until a space error tolerance is reached.

In contrast to this previous work, we seek to maximally exploit frame-to-frame coherence with view-dependent refinement, similar to the ROAM algorithm, but with chunked/patch geometry and texture tiles paging in from disk. High-quality low-pass filtering is applied to geometry tiles in addition to textures so as to minimize geometric aliasing artifacts and to reduce average geometric error. A new Sierpinski disk layout improves coherence of tile access and caching, while the 4-8 textures minimize visible seams at patch boundaries. Like ROAM, our algorithm can maintain near-constant frame rates by optimizing to a triangle budget in addition to selecting a desired screen error tolerance.

3 The Diamond Data Structure

Underlying all the work in this paper is the notion of a *diamond*, which is uniquely associated with one vertex, one edge, and one quadrilateral face in a 4-8 mesh hierarchy. Figure 3 depicts a diamond d with a standard orientation and labeling of its ancestors $a_{0\ldots3}$ and children $c_{0\ldots3}$. By a *parent* of diamond d we mean a diamond one level coarser in the 4-8 mesh whose area overlaps d. Similarly, a *child* of d is one level finer and overlaps d.

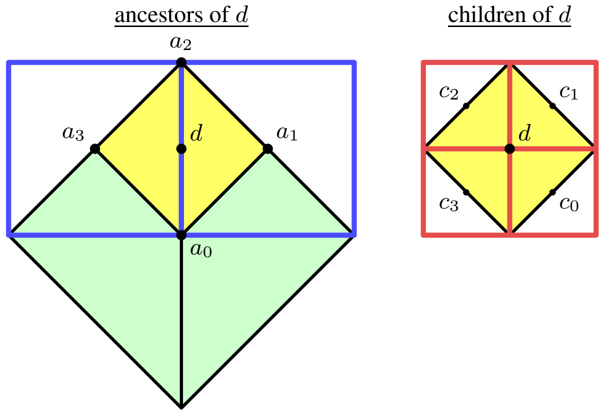

Figure 3: A diamond d (yellow) is shown with respect to its ancestors (left) and its children (right). By numbering each of these counterclockwise around d, and by placing the quadtree ancestor (green) as a_0, and the first child c_0 just after this, navigation through the 4-8 mesh becomes straightforward. Note that the two parent diamonds (blue outline) are the right parent, a_1, and the left parent, a_3. The children of d are $c_{0\ldots3}$, outlined in red.

After experimenting with a number of implementations of 4-8 mesh data structures that support selective refinement, including pointer-free "pure index" schemes, we found after performance profiling that the fastest choice is simply to keep pointers to the children and ancestors, and allocate diamond records in arrays of several thousand at a time to avoid per-record heap allocation overhead. Navigation to a diamond's parent, quadtree and older corner ancestors, as well as children, is then a matter of following single links, which will be denoted $d \rightarrow a_i$ and $d \rightarrow c_i$ respectively for $i = 0 \ldots 3$. Traversing to neighbors at the same level of resolution turns out to be simple as well.

To get to diamond d's neighbor d_0 across the child $d \rightarrow c_0$ edge, Figure 4 shows that both d and d_0 are children of d's right parent $d \rightarrow a_1$. Indeed, d_0 is the child of $d \rightarrow a_1$ just counterclockwise of d. Since moving to neighbors is a frequent operation, it can improve performance to store d's index as a child with respect to both parent a_1 and a_3; these indices will be referred to as $d \rightarrow i_1$ and $d \rightarrow i_3$, respectively. This means that the assertion $d = d \rightarrow a_1 \rightarrow c_{d\rightarrow i_1}$ should always hold for the right parent, and similarly using a_3 and i_3 for the left parent. The pseudocode for moving to the c_0 neighbor of d is then simply

$$\begin{array}{rcl} i & \Leftarrow & (d \rightarrow i_1 + 1) \bmod 4 \\ d_0 & \Leftarrow & d \rightarrow a_1 \rightarrow c_i \end{array}$$

Child edges $d \rightarrow c_{1\ldots3}$ are treated similarly.

Now that neighbor-finding is established, the process of adding a child diamond, say $c = d \rightarrow c_0$, is a matter of finding the neighbor d_0 as above, which is the other parent of c. If d_0 is missing, then it should be recursively added to its parent $d \rightarrow a_1$ at the expected child index. To hook up c properly, first note that its quadtree ancestor $c \rightarrow a_0$ is $d \rightarrow a_1$, the mutual parent of c's two parents d and d_0. This determines the exact orientation of c (just rotate Figure 4 135° clockwise), and thus indicates how all of its ancestors should be filled in, as well as its parent's back pointers:

$$\begin{array}{rcl@{\qquad\qquad}rcl} c \rightarrow a_0 & \Leftarrow & d \rightarrow a_1 & d \rightarrow c_0 & \Leftarrow & c \\ c \rightarrow a_1 & \Leftarrow & d & c \rightarrow i_1 & \Leftarrow & 0 \\ c \rightarrow a_2 & \Leftarrow & d \rightarrow a_0 & d_0 \rightarrow c_3 & \Leftarrow & c \\ c \rightarrow a_3 & \Leftarrow & d_0 & c \rightarrow i_3 & \Leftarrow & 3 \end{array}$$

The last two assignments follow from the observation that d and d_0 both have $d \rightarrow a_0$ as their quadtree ancestor. As before, similar procedures exist for creating children $c_{1\ldots3}$ of diamond d.

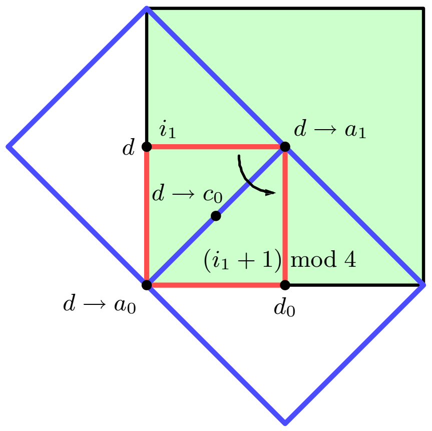

Figure 4: The neighbor of diamond d across its child c_0 edge, d_0, is obtained by walking up from d to its right parent $d \to a_1$, and then d_0 is this parent's child counterclockwise one step from d. To make this computation fast, d's child index within a_1 is kept in d's record, and the counterclockwise child index is this index plus one, taken mod 4.

To delete a childless diamond d, the pointers to d from its parents must be cleared:

$$\begin{array}{lcl} d \to a_1 \to c_{d \to i_1} & \Leftarrow & \text{null} \\ d \to a_3 \to c_{d \to i_3} & \Leftarrow & \text{null} \end{array}$$

Any adaptive 4-8 mesh may be constructed by sequences of child additions and childless-diamond deletions. Convenience operations, such as deleting a diamond with children, may be implemented easily using these basic operations.

The final idea required to begin using diamond meshes is the method to hook up the initial base (i.e. coarsest-level) mesh. Given any manifold polygonal mesh, a diamond base mesh may be constructed by creating a diamond per vertex, face and edge. Vertex diamonds exist only to supply their centerpoint coordinate—no use is made of their child or ancestor links. Face diamonds link to their children, which are the edge diamonds. Conversely, the edge diamonds link to their parents, the face diamonds, as well as their other ancestors, which are vertex diamonds. For polygonal meshes with non-quadrilateral faces, the number of children of face diamonds will not be four, and neighbor-finding will require arithmetic modulo the number of edges in the face. Indeed, the neighbor of d (*e.g.* d_0) in the child-addition procedure may need to examine which of its parents is in common with d in order to select its appropriate child index. In contrast, for the non-base-mesh case of Figure 4, and for cubical base meshes laid out carefully, d_0 always uses child index 3. For this reason, we choose a cubical base mesh for planetary geometry, which has all quadrilateral faces.

The proper layout for a base cube divides the edge diamonds into four sets of three, as shown in Figure 5, with each 3-set sharing a common vertex diamond as their "quadtree" ancestor.

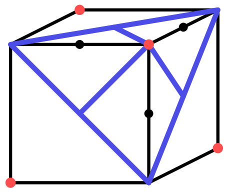

Figure 5: For planetary base meshes, a cube is used, with diamonds for each vertex, face and edge. The edge diamonds should be oriented as shown, so that their a_0 (quadtree) ancestors are one of the four red vertex diamonds, and the face diamonds are their parents. Three edge diamonds sharing the centermost vertex diamond are highlighted in blue.

4 Geometry and Texture Tiles

Given the basic diamond structures just outlined, it is possible to create selectively-refinable objects by associating spatial coordinates and colors to the vertex of each diamond. However, this kind of fine-grained treatment of geometry and color is very inefficient for paging from disk and for rendering on newer graphics hardware. To overcome this, small regular grids of points and colors, called *tiles*, will be associated with each diamond. The central ideas required to work with tiles are to:

1. set up a parametric coordinate system within a diamond, and determine the mapping from child to parent diamond parameters,
2. perform low-pass filtering to create high-quality coarsened tiles, and
3. create additional detail through 4-8 subdivision and optional procedural displacements.

For each diamond, define its local coordinate system $(u, v) \in [0, 1]^2$ to have its origin at the quadtree ancestor vertex $d \to a_0$, the u axis moving from the origin to the right parent $d \to a_1$, and the v axis moving from the origin to the left parent $d \to a_3$. A diamond d overlaps one half of each of its children, in the shape of a right isosceles triangle. The relationship between d's (u, v) coordinates and those in each child is depicted in Figure 6.

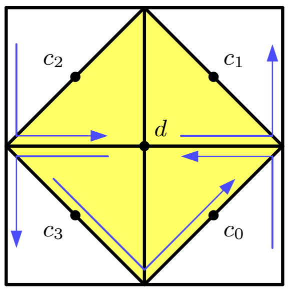

Figure 6: The mapping of diamond (u, v) parameters between a diamond d and its children is depicted using arrows to indicate the u axes. These coordinate systems are standardized to be right-handed, with the origin at the quadtree ancestor vertex. Each diamond's parametric coordinates are in the unit square, that is, $(u, v) \in [0, 1]^2$.

To move information from finer to coarser tiles for low-pass filtering, the tile for d must collect information from half of each child. An affine mapping from child c_i's parameters (u_i, v_i) to d's parameters (u, v) would then be

$$(u, v) = (u_c, v_c) + u_i(u_a, v_a) + v_i(-v_a, u_a)$$

where the origin (u_c, v_c) and u_i direction vector (u_a, v_a) are given in Table 1. These child-to-parent mappings may be composed together to map to coarser ancestors, a process which will be used to obtain texture coordinates in section 7.

child	(u_c, v_c)	(u_a, v_a)
c_0	$(1, 0)$	$(-\frac{1}{2}, \frac{1}{2})$
c_1	$(1, 0)$	$(\frac{1}{2}, \frac{1}{2})$
c_2	$(0, 1)$	$(\frac{1}{2}, -\frac{1}{2})$
c_3	$(0, 1)$	$(-\frac{1}{2}, -\frac{1}{2})$

Table 1: Origin and u_i axis for child-to-parent mappings.

Low-pass filtering for diamond d can now be defined as collecting tile array entries from the appropriate half of each of the four children, and placing these into two arrays arranged according to the local coordinate system of d. As shown in Figure 7, one set of values will be the cell-centered entries (hollow dots), while the other values are vertex centered (solid dots). The new vertex-centered values will be stored in d's tile, and are computed using weighted averages of the old cell- and vertex-centered values obtained from the children. Note that for the weighting mask chosen, there are four cell-centered values (each marked with an X) that are needed, but are outside those available from the four children. While it is possible to query four additional tiles to obtain these values, only a single value from each tile would be used, and has only a tiny impact on quality. Therefore we choose instead to use a slightly altered weight mask for the four corners of d. For geometry tiles to avoid cracks on patch boundaries, section 6 discusses which parent values must be subsamples (simple copies) of the vertex-centered values from the children.

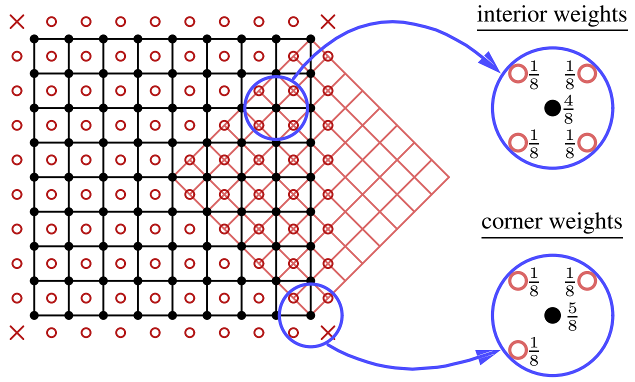

Figure 7: Low-pass filtering is performed by collecting both cell-centered values (hollow dots) and vertex-centered values (solid dots) from the four children of a diamond. One child is highlighted, and the weight masks for the interior and corner cases are given.

Performing 4-8 mesh refinement with tiles is very similar to low-pass filtering, only performed in reverse. The main difference is that a new diamond child tile must collect values from its two parents, and for subdivision schemes smoother than linear or bilinear interpolation, ghost values are needed.

5 Diamond Sierpinski Indices and Paging

When accessing a large terrain database from disk during interaction, performance is highly sensitive to the spatial coherence of the data layout, and is improved by the use of hierarchical space-filling curves [14]. With the kind of tile-based, explicit paging scheme that we are pursuing, we need a fast and local means of mapping diamonds to indices that provides such a good layout, and works well with incremental selective refinement (i.e. diamond child additions and deletions driven by dual priority queues). The most natural and coherent of the space-filling curves to apply to 4-8 meshes is the Sierpinski curve, depicted in Figure 8. Recall from Knuth [10] that any complete binary tree may be assigned unique indices by setting the root node to 1, and then for every node with index k, recursively set it's child indices to be $2k$ and $2k+1$ respectively. Performing this for the triangle bintree gives the indices shown (note that left branches are taken first on even levels, and right branches first on odd levels).

A challenge with these Sierpinski indices is that they are associated with the triangles of a 4-8 mesh, not the diamonds (or equivalently, the vertices). The most obvious choice, associating the index with the triangle's split point, creates two indices per diamond. Associating with any of the three corners results in even worse duplication. It turns out that associating the triangle's index with one

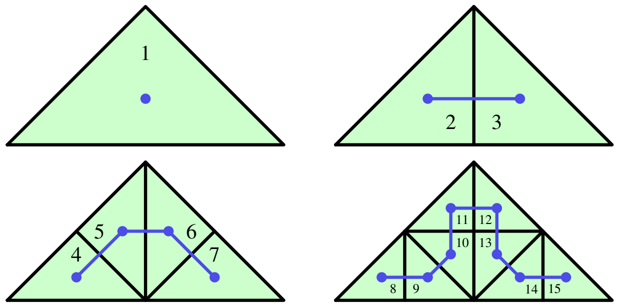

Figure 8: Sierpinski indices for bintree triangles are computed recursively from their parent index. While the layout is highly coherent, the indices are mapped to triangles, not diamonds.

of the midpoints of the shorter edges, say the left side, provides the one-to-one and onto mapping that is needed. Figure 9 provides a visual proof that all diamonds at a given level of resolution are covered exactly once by the left edges of bintree triangles one level coarser in the 4-8 mesh.

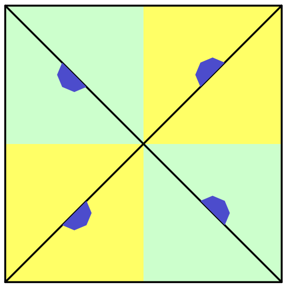

Figure 9: The 4-edge neighborhood shown is covered exactly once by the diamonds associated with the left edges of the bintree triangles. This pattern repeats to cover the plane. The triangles are shown in outline, the diamond areas in alternating shades, and the diamond centers by marking the inside of their respective bintree triangle left edge.

To compute the Sierpinski index of a diamond d efficiently during selective refinement, the diamond must be mapped to its Sierpinski triangle, namely the bintree triangle whose left edge has the diamond vertex at its center. From this Sierpinski triangle, its parent Sierpinski triangle is determined, and then the diamond of its left edge is the "Sierpinski parent" d_S of d. There are two cases, as shown in Figure 10, depending on whether the distinguished diagonal of d's quadtree parent $d \rightarrow a_0$ is horizontal or vertical:

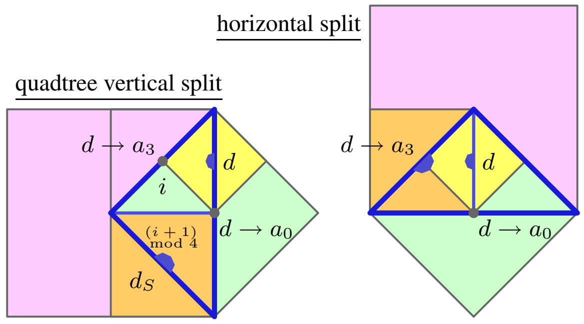

Figure 10: The Sierpinski parent d_S of a diamond d is determined based on two cases, depending on the orientation of d's quadtree ancestor's distinguished edge. On the left, this edge is vertical, and the counterclockwise neighbor of d's left parent is the Sierpinski parent. On the right, the Sierpinski parent is simply d's left parent.

The pseudocode to compute d's Sierpinski index $d \rightarrow k$ is then:

```
$d_3 \Leftarrow d \rightarrow a_3$
if $d_3 \rightarrow a_1 = d \rightarrow a_0$, then
    $d_S \Leftarrow d_3 \rightarrow a_1 \rightarrow c_{(d_3 \rightarrow i_1 + 1) \bmod 4}$   *create as needed*
    $d \rightarrow k \Leftarrow 2 d_S \rightarrow k + x$
otherwise
    $d_S \Leftarrow d_3$
    $d \rightarrow k \Leftarrow 2 d_S \rightarrow k + y$
```

where for even levels of the 4-8 mesh, $(x, y) = (1, 0)$, and for odd levels $(x, y) = (0, 1)$.

A diamond's index is stored in 64-bits, where the upper bits represent the Sierpinski index followed by a one and a string of zeros to the end. To map a Sierpinski index to input and output of files, blocks and tiles, we consider a Sierpinski index to be left-shifted so that the leading "1" bit is just removed in a 64-bit register, and place that bit just to the right of the least significant bit of the index in order to mark the end of the relevant bits:

```
$i \Leftarrow (i \ll 1) | 1$
$\mathrm{MSB} = 1 \ll 63$
while ( $(i \& \mathrm{MSB}) = 0$ ) $i \Leftarrow i \ll 1$
$i \Leftarrow i \ll 1$
```

The bits are now of the following form:

$b_{63} b_{62} b_{61} ... b_N 100...0$

where N is the least significant bit of the Sierpinski index after the left-shift procedure.

This bit string can now be treated like a generalized directory path name, at first literally describing directory branches, then a file name, followed by the block index and tile number within the block. We explain using the case $N = 37$:

$b_{63} b_{62} b_{61} b_{60}$ } directory branch 1

$b_{59} b_{58} b_{57} b_{56}$ } directory branch 2

$b_{55} b_{54} b_{53} b_{52}$ } directory branch 3

$b_{51} b_{50} b_{49} b_{48}$ } file name

$b_{47} b_{46} b_{45} b_{44} b_{43} b_{42} b_{41} b_{40}$ } block number within file

$b_{39} b_{38} b_{37} 10$ } tile number within block

The "1" mark bit is allowed to be in any of the five tile bit positions. A special root file is made in the top-level directory to catch all the blocks and tiles that have insufficient bits to define a full 16-bit file index. This leads to directories with up to 16 subdirectories and 16 files each, where each file contains up to 256 read/write blocks, each of which contains up to 32 tiles from 5 different levels of detail. Branching factors, block sizes and so on can be tuned for performance, but we found the arrangement given here to be very effective on the systems we tested.

When a tile is requested, it is returned immediately if it is in main memory. If it is in a compressed read/write block in memory, the tile is decompressed and placed in the tile cache. If the block is missing from the cache, it is read into the block cache from disk, and the tile is extracted. If this process fails to find a tile, the tile is manufactured using 4-8 subdivision and optional procedural displacements. Since elevation and texture tiles are simple 2D rasters, any number of known compression schemes can be applied.

For this system we use a least-recently-used strategy for tile and block cache replacement decisions. Cache sizes should be determined by balancing various application and system memory needs, since of course there is incremental gain for any increase in a particular cache as long as another cache is not decreased. For our system, we found a total cache size of a hundred megabytes, divided evenly between compressed-tile blocks and uncompressed tiles, provides excellent performance.

6 Geometry Patches and Frame-to-Frame Updates

When replacing individual leaf triangles with small patches of say 1024 triangles, a natural concern is that a loss of adaptivity will result. However, modern graphics hardware can render thousands of such patches at 50-100 frames per second, which is similar to the performance for thousands of *single* triangles reported for view-dependent HRT algorithms less than a decade ago.

From [18], we know that for any uniform refinement of a right isosceles triangle that is a power of four, such as 256 or 1024, the patches of an adaptive 4-8 mesh will be without cracks. For most efficient rendering, these patches are laid out as vertex and indexed-triangle arrays, where both the vertices and triangles are listed in Sierpinski order, as shown in Figure 11 for the case of 256 triangles per patch. Note that the 256-triangle patch has 16 triangle edges per patch edge, thus ensuring crack-free selective refinement.

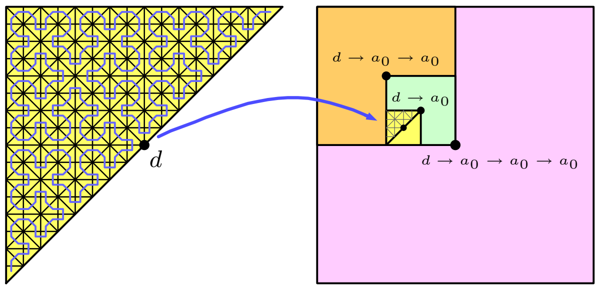
$d \rightarrow a_0 \rightarrow a_0$
$d \rightarrow a_0$
d
$d \rightarrow a_0 \rightarrow a_0 \rightarrow a_0$

Figure 11: The Sierpinski layout of a triangle patch, with the mapping of the patch to its elevation tile. If d is the diamond of the triangle patch, then the child-to-parent mappings of section 4 can be composed to locate the appropriate elevation values in the third quadtree ancestor, $d \rightarrow a_0 \rightarrow a_0 \rightarrow a_0$.

For geometry, the triangular patches are best taken as only a small fraction of a CPU-cache tile, since the optimal granularity of these two objects is quite different. After testing a number of sizes, we found a good tradeoff to be a tile with 129 or 257 vertices (elevation samples) per side. For triangular patches, either 256 or 1024 triangles are used. Figure 11 shows a 256-triangle patch in relation to a tile with 129×129 vertices. Note that for these sizes, the tile diamond is the third quadtree ancestor of the patches' diamond.

The low-pass filtering scheme from section 4 is used for elevation tiles, but with some vertices being subsampled to avoid creating cracks during selective refinement. It is sufficient to subsample the vertices on the edges of the patch diamonds, and allow their interiors to be smoothed out through low-pass filtering. For example, in Figure 11, the four sides of diamond d (in yellow) should be subsampled.

Frustum culling for triangle patches is identical to the system used in ROAM, but we simplify the method to use bounding spheres rather than pie-wedge bounds, thus reducing by about six the per-plane floating-point frustum in/out tests. In addition, since the core data structure is now a diamond rather than a bintree triangle, it is natural to pass frustum-cull in/out flags down from the quadtree ancestor, which have a nesting relationship, rather than the parent, which doesn't. We can avoid getting overly-conservative culling by indicating a triangular patch is out if either its diamond is out, or the parent diamond on that patches' side is out. As with ROAM, entire subtrees of in/out labels will remain constant from frame to frame if its root diamond stays either out or all in from the previous to the current frames, and hence no subtree work is needed.

Similar to ROAM, dual-queues are used to prioritize respectively diamond split and merge activity. Unlike the ROAM base priority that is sensitive to surface roughness, we only use the estimated screen size of the diamond as its split/merge priority, so as to perform geometric antialiasing given the extremely high triangle counts available.

7 4-8 Textures for Triangle Patches

Most multi-resolution texture algorithms use a prefiltered quad-tree of textures, where tiles all have the same number of texels but where quadtree children cover one fourth the area of their parent. Selecting adjacent tiles where the texels per unit area differ by a factor of four can produce visual discontinuities. Our method creates twice as many detail levels, allowing a smoother transition between levels (only factors of two), while effectively using the diamond hierarchy for level traversal.

The initial data set texture is diced into 128^2 or 256^2 size tiles, which represent the texture at the finest level. Low-pass filtering is performed as described in section 4. The filtering approach from level-to-level preserves the average energy of the original signal to minimize level-of-detail transition artifacts. Unlike geometry filtering, which must subsample on the boundaries of patch diamonds, texture tiles appear more visually seamless without any subsampling (subsampling can alter the average energy near boundaries, thus producing visual artifacts).

Each displayed triangle patch is evaluated to determine its optimal texture resolution. Since patches are drawn using a single rendering call, no more than one texture tile can be associated with a triangle patch. Hence the finest resolution texture that can be accessed will be at the same diamond level as a patches' diamond. For a 128^2 texture tile and a 256-triangle patch, this means a maximum of 32 texels per triangle. Since graphics hardware will exhibit differences in relative texel and triangle rendering performance, we decouple the geometry and texture levels of detail. For high triangle performance relative to texture performance or memory availability, fewer than 32 texels per pixel may be desired. Ideally if texture performance were not a bottleneck we would choose a texel-to-pixel ratio near one, and determine the texture level of detail using this. Using the child-to-parent parameter mapping from section 4, one can iteratively walk to the diamond parent on the side containing the triangle patch until the desired texture level is reached. The texture coordinates for the patch vertices can then be easily computed using the resulting composite mapping.

Using the bounding sphere radius previously calculated for frustum culling, we compute an upper bound on the possible screen area covered by the triangle-patch diamond. The maximum screen space coverage occurs when looking at a diamond oriented perpendicular to the view direction. We use as the upper bound on pixel area $2R^2$, where R is the projected radius of the diamond's bounding sphere. Using the number of texels in the texture diamond covered by the triangle patch, the texel-to-pixel ratio α is computed. Frame-to-frame, the patch-to-texture level-of-detail associations are adjusted incrementally, similar to the split-merge dual-queues for geometry, so as to keep α close to 1.0. Higher priority is given to refining a patches' texture association as α becomes greater than one, and coarsening becomes more urgent as α becomes less than one. We keep to a budget of $4 - 8$ patch-to-texture updates per frame to maintain high frame rates, since each update can be expensive.

If the desired texture is not cached in texture memory, we use the next coarser texture level that is available. When finer textures are loaded, we keep coarser textures so that the system can always instantly coarsen as desired. The next finer texture diamond is then added to a texture-wait queue with priorities defined by the α of this next-finer texture. Because updates to texture memory are expensive, the wait queue allows a fixed number of textures to be uploaded per frame, thus avoiding irregular load times. When a texture is to be cached, it is fetched from the disk tiles using the diamond's 64-bit index, as described in section 5.

Each triangle now has a cached texture associated with it. If the level-of-detail for a triangle has changed or the texture has just been cached, we must compute the new texture coordinates for the triangle patch, using the composite child-to-parent mappings.

8 Results

Our performance results were measured using a 3Ghz Xeon processor with 1GB of RAM and a GeForce FX 5900 Ultra. We ran the tests at a resolution of 640×480 utilizing the NVidia vertex array range specification combined with chunked triangle patches to exploit the graphics-card capabilities. These results are based on a flight path through the 10-meter data of Washington state [19] with around 1.4 billion elevation and texel values at the finest resolution. The source elevation data totals 2.7 gigabytes on disk before preprocessing. Textures were procedurally generated and colored from the original geometry and stored in RGB-565 format.

The out-of-core preprocessing step for this particular data set took approximately 53 minutes including the calculation of the shaded texture map from the geometry. Without the shading step, preprocessing texture and geometry data into tiles took 33 minutes.

In the rendering application, approximately 53% of the time for a given frame is spent preparing the vertex array data. During this time, vertex pointers are set up and triangle patches that need to be updated either due to geometry updates or texture coordinate updates are transferred to AGP memory to be pulled by the GPU. Around 45% is spent managing vertex and texture coordinate cache allocation and traversing the hierarchy to evaluate when triangle patches or texture coordinate updates are necessary. The time taken by the split/merge optimization loop is a user defined parameter, but in this test less than two percent time was spent on this. Less than one percent each was spent on fetching geometry and texture from disk, priority updates, coordinate mapping calculations, triangle patch building, frustrum culling, and new texture loading. In our implementation, priority queues also allowed a user-defined number of fixed textures to be sent to graphics-card texture memory per frame. Our results show that the main bottleneck lies in the graphics-card upload bandwidth and the loop for determining appropriate triangle patch updates to geometry and texture.

Performance statistics for our implementation are shown in Figure 12, taken during a flyover of Mount Rainier (shown in the accompanying video). In the lower-right graph, the rendering preparation line refers to the updating of AGP memory and set up of vertex pointers. Traversal and allocation involves walking through the diamond hierarchy and managing system memory. The geometry optimization line represents the split/merge time taken per frame. The remaining calculations, generally taking less than two percent of the frame time, are labeled "other". Snapshots from the flyover are highlighted in Figure 13.

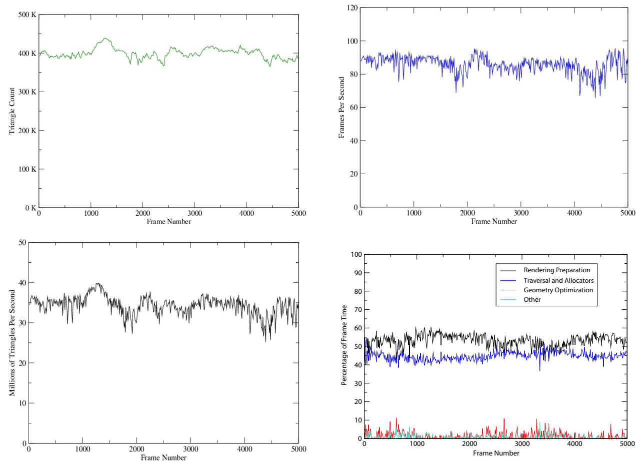

Figure 12: Performance graphs measured for a test flight path over the 10-meter WA state data: (top left) near-constant triangle counts matching the triangle budget target, (top right) frames per second, (bottom left) Mtri per second, and (bottom right) % breakdown of system task times.

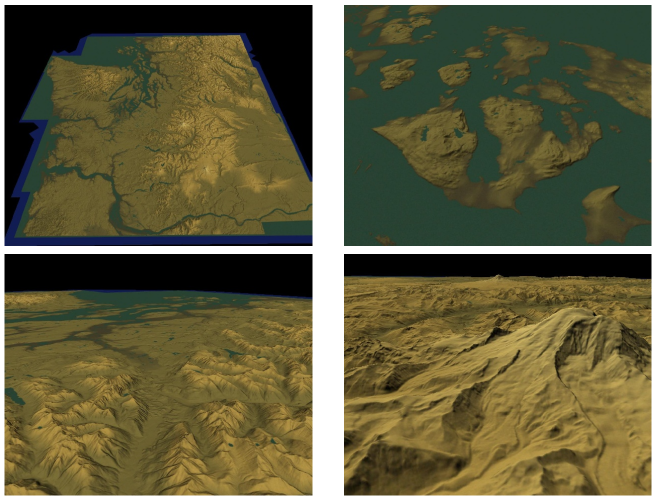
Figure 13: Screen shots of our test flight showing the overall Washington state data set, the San Juan islands, a view facing Victoria, and Mount Rainier with Mount Adams behind.

9 Conclusion

We have presented a solution to the texture level-of-detail problem for real-time view-dependent rendering of extremely large terrain meshes. We introduce a new texture hierarchy based upon a 4-8 mesh, which, when coupled with a similar adaptive geometry scheme, provides a mechanism for real-time display of the terrain. The 4-8 hierarchy provides twice as many levels of detail as conventional quadtree-style refinement schemes such as mipmaps. Because of this more gradual change, we find in practice that the transitions between texture levels of detail are less perceptible. The 4-8 scheme is integrated into a variant of the ROAM algorithm, and together with a simple out-of-core data access mechanism based upon Sierpinski curves allows out-of-core access for the display of very large textured meshes.

Future work based on this terrain system can be expanded to include dual queues at all levels of cache, for both geometry and texture, replacing the reactive least-recently-used strategy with prefetching and optimized priority modeling. Anisotropic filtering could help with highly warped terrain data, such as near cliffs, and with the horizon aliasing for near-planar regions. Further experimentation with different types of texture maps, such as normal maps for lighting calculations, may enhance the visual quality of a scene and allow dynamic lighting. As memory bandwidth increases, it may also be possible to play animated textures of certain areas in a scene to demonstrate time varying properties like plant life or erosion. The development of realtime, high quality procedural detail is also of interest.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. This work was supported by the National Science Foundation under contracts ACR 9982251 and ACR 0222909, through the National Partnership for Advanced Computing Infrastructure (NPACI), and by Lawrence Livermore National Laboratory under contract B523818. We thank the members of the Visualization and Graphics Group of the Institute for Data Analysis and Visualization (IDAV) at UC Davis, and the visualization researchers within the Center for Applied Scientific Computing at LLNL.

References

[1] National Aeronautical and Space Administration. MOLA data set, http://pds-geosciences.wustl.edu/missions/mgs/megdr.html, 2003.

[2] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. BDAM: Batched dynamic adaptive meshes for high performance terrain visualization. In *Proc. EG2003*, pages 505–514, September 2003.

[3] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Interactive out-of-core visualization of very large landscapes on commodity graphics platforms. In *ICVS 2003*, pages 21–29. November 2003.

[4] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet–sized batched dynamic adaptive meshes (P-BDAM). In *Proc. IEEE Visualization*, pages 147–155, October 2003.

[5] Jürgen Döllner, Konstantin Baumann, and Klaus Hinrichs. Texturing techniques for terrain visualization. In *Proc. IEEE Visualization*, pages 227–234, 2000.

[6] Mark A. Duchaineau, Murray Wolinshy, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *Proc. IEEE Visualization*, pages 81–88, October 19–24 1997.

[7] William Evans, David Kirkpatrick, and Gregg Townsend. Right triangular irregular networks. Technical Report TR97-09, The Department of Computer Science, University of Arizona, May 1997.

[8] William Evans, David Kirkpatrick, and Gregg Townsend. Right-triangulated irregular networks. *Algorithmica*, 30, 2001.

[9] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proc. IEEE Visualization*, 1998.

[10] D. E. Knuth. *The Art of Computer Programming, Sorting and Searching*. 2nd edition, 1975.

[11] Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *Proc. IEEE Visualization*, pages 259–266, October 2002.

[12] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hughes, Nick Faust, and Gregory Turner. Real-Time, continuous level of detail rendering of height fields. In *SIGGRAPH 96 Conference Proceedings*, pages 109–118, August 1996.

[13] Peter Lindstrom and Valerio Pascucci. Visualization of large terrains made easy. In *Proc. IEEE Visualization*, pages 363–370, 2001.

[14] Peter Lindstrom and Valerio Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, July/September 2002.

[15] Anthony Mirante and Nicholas Weingarten. The radial sweep algorithm for constructing triangulated irregular networks. *IEEE Computer Graphics and Applications*, 2(3):11–13, 15–21, May 1982.

[16] Tomas Möller and Eric Haines. *Real-Time Rendering*. A. K. Peters Limited, 2nd edition, 1999.

[17] Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proc. IEEE Visualization '98*, pages 19–26,515, 1998.

[18] Alex Pomeranz. ROAM using surface triangle clusters (RUSTiC). M.S. thesis, Department of Computer Science, University of California, Davis, June 2000.

[19] United States Geological Service. State of Washington data set, http://rocky.ess.washington.edu/data/raster/tenmeter/onebytwo10/index.html.

[20] Cláudio T. Silva, Joseph S. B. Mitchell, and Arie E. Kaufman. Automatic generation of triangular irregular networks using greedy cuts. In *Proc. IEEE Visualization*, pages 201–208, 1995.

[21] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: A virtual mipmap. In *SIGGRAPH 98 Conference Proceedings*, pages 151–158, July 1998.

[22] Thatcher Ulrich. Rendering massive terrains using chunked level of detail control, SIGGRAPH Course Notes, 2002.

[23] Luiz Velho. Using semi-regular 4-8 meshes for subdivision surfaces. *Journal of Graphics Tools*, 5(3):35–47, 2000.

[24] Lance Williams. Pyramidal parametrics. *SIGGRAPH '83 Proceedings*, 17(3):1–11, July 1983.