Title
Real-Time Optimal Adaptation for Planetary Geometry and Texture: 4-8 Tile Hierarchies

Journal
IEEE Transactions on Visualization and Computer Graphics, 11

Authors
Hwa, Lok Ming
Duchaineau, Mark A.
Joy, Ken

Publication Date
2005

Peer reviewed

Real-Time Optimal Adaptation for Planetary Geometry and Texture: 4-8 Tile Hierarchies

Lok M. Hwa, Mark A. Duchaineau, *Member*, *IEEE*, and Kenneth I. Joy, *Member*, *IEEE*

Abstract—The real-time display of huge geometry and imagery databases involves view-dependent approximations, typically through the use of precomputed hierarchies that are selectively refined at runtime. A classic motivating problem is terrain visualization in which planetary databases involving billions of elevation and color values are displayed on PC graphics hardware at high frame rates. This paper introduces a new *diamond* data structure for the basic selective-refinement processing, which is a streamlined method of representing the well-known hierarchies of right triangles that have enjoyed much success in real-time, view-dependent terrain display. Regular-grid tiles are proposed as the payload data per diamond for both geometry and texture. The use of 4-8 grid refinement and coarsening schemes allows level-of-detail transitions that are twice as gradual as traditional quadtree-based hierarchies, as well as very high-quality low-pass filtering compared to subsampling-based hierarchies. An out-of-core storage organization is introduced based on Sierpinski indices per diamond, along with a tile preprocessing framework based on fine-to-coarse, same-level, and coarse-to-fine gathering operations. To attain optimal frame-to-frame coherence and processing-order priorities, dual split and merge queues are developed similar to the Realtime Optimally Adapting Meshes (ROAM) Algorithm, as well as an adaptation of the ROAM frustum culling technique. Example applications of lake-detection and procedural terrain generation demonstrate the flexibility of the tile processing framework.

Index Terms—Large data set visualization, level-of-detail techniques, view-dependent visualization, adaptive textures, out-of-core algorithms, procedural terrain generation.

1 Introduction

Planetary data sets are readily available with over a billion elevation and color values [1], [27]. Displaying good approximations of these databases on PC hardware at high frame rates is an ongoing challenge as the sizes of the databases and the opportunities afforded by new graphics hardware both grow. We consider the case when geometry and color data is too extensive to fit in core, but must be paged from disk both during pre-interaction processing (hierarchy building and the like), as well as during interactive display. Given that the databases are hundreds to thousands of times as large as can be displayed at high frame rates, reductions in complexity are needed. Ideally, geometry and texture approximations should be optimized dynamically based on the viewpoint. Historically, view-dependent optimizers worked at a fine-grained level, adding or subtracting only two triangles at a time. This makes it hard to exploit newer graphics hardware that works best with rendering units that consist of larger collections of triangles. The ROAM Algorithm [8], which makes ideal use of frame-to-frame coherence to prioritize coarsening and refinement work, is used as the basis in this paper for ordering selective-refinement operations and frustum culling. This basic scheme is enhanced with a streamlined data structure, out-of-core indexing, and a tile-processing system. This coarse-grained selective refinement of geometry and images is made more seamless at level-to-level transitions through the use of 4-8 tile hierarchies that have very gradual level-of-detail changes and through the use of high-quality low-pass filtering. An example of a one-meter database of Fort Hunter Liggett, California, shown in Fig. 1, demonstrates how seamless these transitions can be, even without the use of per-pixel blending (mipmaps) to hide the seams.

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 nonuniform in screen space due to variations in surface roughness since this will only lead to subpixel 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

- *L.M. Hwa can be reached at 607 Westview Place, Chula Vista, CA 91910. E-mail: lok.hwa@gmail.com.*
- *M. Duchaineau is with the Lawrence Livermore National Laboratory, 7000 East Ave. L-557, Livermore, CA 94551. E-mail: duchaine@llnl.gov.*
- *K.I. Joy is with the Institute for Data Analysis and Visualization and the Department of Computer Science, University of California, Davis, One Shields Avenue, Davis, CA 95616-8562. E-mail: joy@cs.ucdavis.edu.*

Manuscript received 9 Oct. 2004; revised 7 Feb. 2005; accepted 16 Feb. 2005; published online 10 May 2005.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-0127-1004.

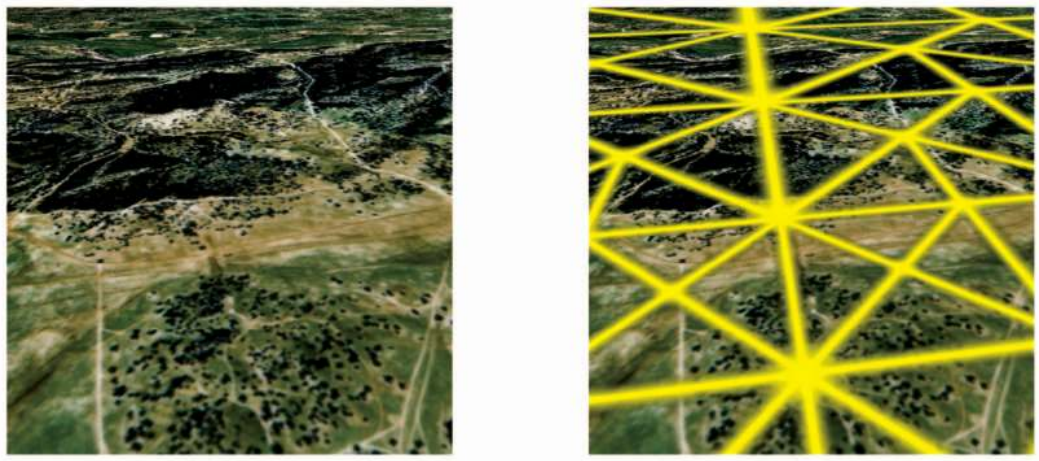

Fig. 1. Two screen shots of an overflight of Fort Hunter Liggett, California, that illustrate the use of 4-8 texture hierarchies. On the left is the seamless textured image produced by the system, while the right shows the outline of the texture tiles used in producing the image.

consideration of texture atlas schemes in which a power-of-two tile is filled with irregular subregions 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 [10], [11]. We chose the 4-8 meshes and their *diamond* elements, shown in Fig. 2, since these match the constraints of texture hardware and have many known desirable properties [19], [8], [20]. At display time, triangle patches associated with the leaf diamonds are drawn, where each patch associates with the most appropriate available texture diamond.

The system proposed here includes two main parts: First, a pre-interaction preparation phase which converts raw input elevations and colors into processed and filtered tile hierarchies on disk and, second, a runtime view-dependent optimization and rendering algorithm that incrementally updates the neighborhoods of geometry or texture tiles. The overall runtime state is depicted in Fig. 3. The goal of this design is to prioritize coarsening and refinement work that is most urgently needed each frame to stay near the target triangle count and texel-to-pixel ratio. The paging and view-dependent optimization states are almost identical for geometry and texture. Both include a hierarchical disk database, caches for compressed I/O blocks and decompressed tile rasters, a selectively refined diamond mesh, and the dual split-merge priority queues that order the incremental updates to the diamond mesh. The block and tile caches use a least-recently-used (LRU) replacement strategy. In addition, geometry patches map to the available texture that is closest to its ideal texel-to-pixel ratio. Since updates to patch-to-texture mappings involve expensive transfers of new texture coordinate arrays to AGP memory, a single priority queue is used to budget these updates per frame. Triangle-patch diamonds and texture-object diamonds are optimized independently using dual queues. Both of these optimization loops are similar to the original ROAM optimization loop for

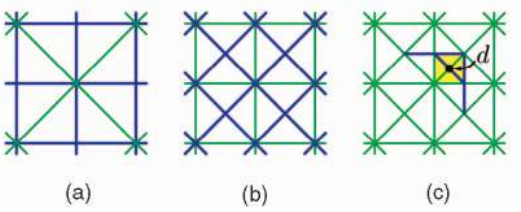

Fig. 2. A 4-8 mesh illustrating different levels of resolution. Part (a) shows a coarse, uniform refinement, which is effectively a grid of squares (blue) with distinguished diagonals (green). Part (b) is one level finer everywhere. Note the blue squares are rotated 45° and scaled by $\sqrt{1/2}$. Part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

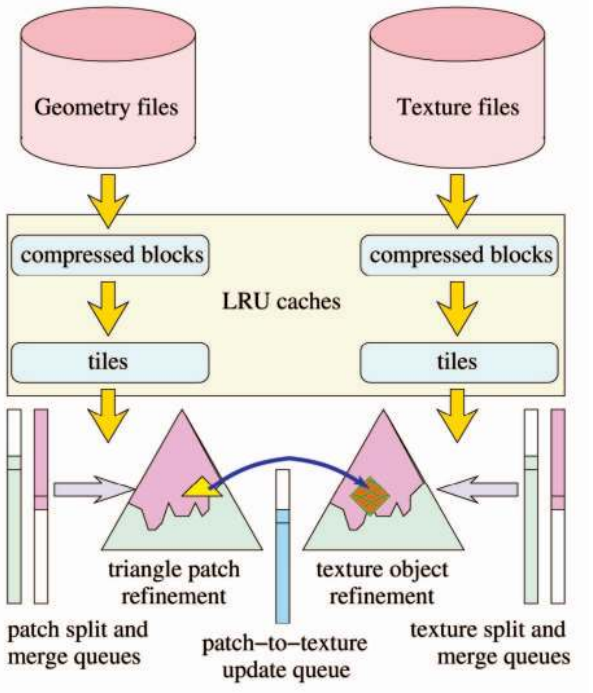

Fig. 3. The system state for frame-to-frame optimization of geometry and texture includes disk hierarchies, least-recently-used caches of blocks and tiles, dual-queue optimizers for patches and texture objects, and a priority queue to budget updates to the patch-to-texture mappings.

triangles. Splits and merges schedule future heavy-weight activity, including the retrieval of data from cache or disk as needed and the uploading of patch or texture data to special graphics-hardware memory. The paging and upload work can be tightly controlled per frame to achieve an application-specific balance of the trade-off between full optimization per frame and fast, even frame rates. The overall optimization and rendering loop per frame is performed in phases as follows:

For each frame {

1. **Update the frustum-cull state and priorities for the active diamonds:**
   - Update frustum for new frame (do not upload to graphics hardware yet).
   - Update frustum-cull IN/OUT labels for all diamonds.
   - Update split/merge priorities for texture and geometry diamonds.
2. **Perform texture-object optimization loop** (uses dual split-merge priority queues). This will schedule heavy-weight operations that will be performed later: access to texture tiles from cache if available and paging from disk as needed. The optimization loop will terminate early to limit the number of paging and texture upload operations per frame.
3. **Perform geometry patch optimization loop** (uses a second dual queue). Geometry tiles will be scheduled to be accessed from cache or paged from disk as needed. The optimization loop will terminate early to limit the number of paging and upload operations per frame.
4. **Determine desired patch-to-texture mappings.** Compute priorities of patch-to-texture mapping updates.
5. **Perform patch-to-texture update loop** (uses a single priority queue). Updates will be scheduled for later and limited to fixed budget per frame.
6. **Swap the display buffers.** Upload new frustum to graphics hardware. Launch rendering of unchanged patches (nonblocking calls).
7. **Perform scheduled paging/upload operations** (concurrent with rendering of unchanged patches).
8. **Launch rendering of updated patches** (this rendering will be performed concurrent with phases 1 through 5).

}

In order to realize this overall optimization and rendering strategy, it is important to understand:

- **The core data structure for diamonds:** While several data structures have been devised to support 4-8 refinement, e.g., [8], [11], [2], 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.
- **Standard diamond parameterizations and level-to-level mappings:** 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 [31] with the optional addition of procedural detail. The basic properties of tiles, including standard orientations and level-to-level mappings, are described in Section 4.
- **Out-of-core indexing using Sierpinski curves:** For efficient input and output, files and disk blocks are laid out using a diamond indexing scheme based on the Sierpinski space-filling curve. Sierpinski indexing and basic data traversals for out-of-core preprocessing are described in Section 5.
- **Filtering operations using diamond raster tiles:** The technique for computing coarser and finer tiles and generally computing local image-processing operations is based on a simple neighborhood-gathering procedure and least-recently-used caching strategy. When combined with tile traversal in Sierpinski order, very efficient and general preprocessing of planetary data can be performed using a small, fixed memory footprint. An extensive sequence of image processing operations, used for automated lake detection, demonstrates the flexibility of this approach. The filtering and caching mechanism for preprocessing is covered in Section 6.
- **Geometry-specific filtering, layouts, and per-frame optimization:** For geometric rendering, *patches* of 256, 1,024, or more 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 [26], [18]. We are able to achieve triangle throughput close to the practical limits on recent PC video cards. Section 7 outlines how patches are laid out and updated via the cache of elevation tiles. Dual priority queues, similar to those of the ROAM algorithm [8], drive the frame-to-frame updates of the adaptive refinement of displayed patches.
- **Texture-specific filtering and optimization:** The adaptive 4-8 textures, defined in detail in Section 8, fill each diamond area with a regular-grid image raster, rendered using bilinear interpolation. Neighboring tiles share boundary samples on their mutual edges. 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. Since this is an expensive operation, updates are budgeted per frame based on a simple priority queue. The texture-object refinement is updated independently from the triangle-patch hierarchy, using a second, similar dual-queue optimizer.

- **Adding procedural detail:** Finally, additional geometric detail is added using a smooth interpolatory subdivision scheme on tiles, combined with random displacements. These operations are shown to be fast enough to adequately feed view-dependent refinement during rapid fly-overs at low altitudes and provides a fair quality procedural terrain. The procedural refinement scheme is covered in Section 9.

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 10, 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 neither per-pixel blending of texture level-of-detail nor per-vertex blending of geometric data seems to be needed; we believe this is largely due to the gradual factor-of-two changes in information content between levels.

2 Related Work

Our previous paper, which introduces the basic diamond data structure and the 4-8 texture hierarchies is [16], and this provides a more detailed review of the related work. An overview of geometric level-of-detail algorithms can be found in [24], where notable categories are Triangulated Irregular Networks (TINs) [13], [28], [15], and Hierarchies of Right Triangles (HRTs) [23], [10], and Nested Regular-Grid (NRG) methods [22]. HRT and NRG methods generally have greater speed and lower memory use, but require more triangles for the same quality [10]. For the reasons outlined earlier, we focus on HRT schemes, such as Lindstrom et al. [19], Duchaineau et al. [8], Lindstrom and Pascucci [20], [21], Gerstner [14], and Pajarola [25].

A number of view-dependent optimizers have been devised that use coarse-grained selective refinement, including an early formal treatment by De Floriani and Puppo [12], Pomeranz's [26] ROAM algorithm extended to use precomputed HRT patches, Levenberg's [18] extensions to allow runtime HRT patch updates, and Cignoni et al.'s [3] use of TIN patches within an HRT region. The basic large-texture processing algorithms are mipmaps [32] and clipmaps [29]. Two algorithms on out-of-core view-dependent geometry are DeCoro andd Pajarola's XFastMesh [6] and El-Sana and Chiang [9].

Methods that combine texture and geometry view-dependent refinement include Ulrich's quadtree-chunk method [30], the multiresolution terrain and clipmap-like texture system of Döllner et al. [7], the geometry plus texture clipmaps of Losasso and Hoppe [22], and the BDAM system of Cignoni et al. [3], [4], [5] that uses HRTs of TIN patches and a quadtree of texture tiles.

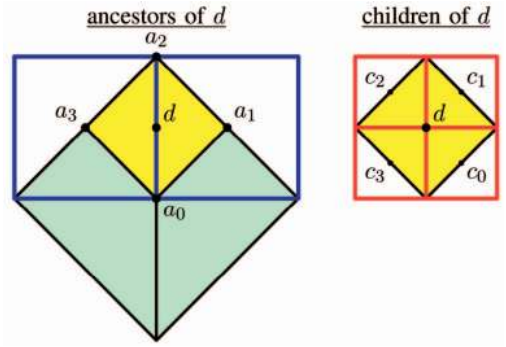

Fig. 4. A diamond d (yellow) is shown with respect to its ancestors (left) and its children (right). By numbering each of these counterclockwise around d and by placing the quadtree ancestor (green) as a_0 and the first child c_0 just after this, navigation through the 4-8 mesh becomes straightforward. 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.

In contrast to this previous work, we seek to maximally exploit frame-to-frame coherence with view-dependent refinement, similarly 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 the 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. Fig. 4 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.

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 ancestor, 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.

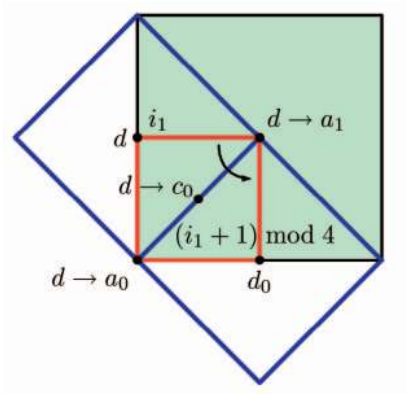

Fig. 5. To get to neighbor d_0 of diamond d across its child c_0 edge, first walk up to its right parent $d \to a_1$. Now, d_0 is the child of a_1 that is one step counterclockwise from d. For faster computation, d's child index i_1 within a_1 is kept in d's record. The counterclockwise child index is $i_1 + 1$, taken mod 4.

To get to diamond d's neighbor d_0 across the child $d \to c_0$ edge, Fig. 5 shows that both d and d_0 are children of d's right parent $d \to a_1$. Indeed, d_0 is the child of $d \to a_1$ 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 $d \to i_1$ and $d \to i_3$, respectively. This means that the assertion $d = d \to a_1 \to c_{d \to i_1}$ 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

$$\begin{aligned} i &\Leftarrow (d \to i_1 + 1) \bmod 4 \\ d_0 &\Leftarrow d \to a_1 \to c_i. \end{aligned}$$

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

Now that neighbor-finding is established, the process of adding a child diamond, say $c = d \to c_0$, begins by 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, $d \to a_1$, at the expected child index. To hook up c properly, first note that its quadtree ancestor $c \to a_0$ is $d \to a_1$, the mutual parent of c's two parents, d and d_0. This determines the exact orientation of c (just rotate Fig. 5 135° clockwise) and thus indicates how all of its ancestors should be filled in, as well as its parent's back pointers:

$$\begin{array}{llll} c \to a_0 \Leftarrow d \to a_1 & d \to c_0 \Leftarrow c \\ c \to a_1 \Leftarrow d & c \to i_1 \Leftarrow 0 \\ c \to a_2 \Leftarrow d \to a_0 & d_0 \to c_3 \Leftarrow c \\ c \to a_3 \Leftarrow d_0 & c \to i_3 \Leftarrow 3. \end{array}$$

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

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

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

Any adaptive 4-8 mesh may be constructed by sequences of child additions and childless-diamond deletions.

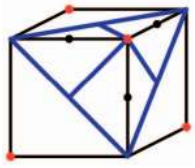

Fig. 6. For planetary base meshes, a cube is used, with diamonds for each vertex, face and edge. The edge diamonds should be oriented as shown so that their a_0 (quadtree) ancestors are one of the four red vertex diamonds and the face diamonds are their parents. Three edge diamonds sharing the centermost vertex diamond are highlighted in blue.

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 nonquadrilateral 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 Fig. 5 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 Fig. 6, with each 3-set sharing a common vertex diamond as their "quadtree" ancestor.

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.

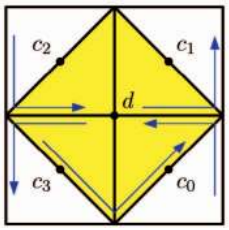

Fig. 7. The mapping of diamond (u, v) parameters between a diamond d and its children is depicted using arrows to indicate the u axes. These coordinate systems are standardized to be right-handed, with the origin at the quadtree ancestor vertex. Each diamond's parametric coordinates are in the unit square, that is, $(u, v) \in [0, 1]^2$.

For each diamond, define its local coordinate system $(u, v) \in [0, 1]^2$ to have its origin at the quadtree ancestor vertex $d \rightarrow a_0$, the u axis moving from the origin to the right parent $d \rightarrow a_1$, and the v axis moving from the origin to the left parent $d \rightarrow a_3$. 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 Fig. 7.

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 8.

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 Fig. 8, 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 (Section 6 avoids the special case on the corners and supports much wider stencils for low-pass filtering). For geometry tiles to avoid cracks on patch boundaries, Section 7 discusses which parent values must be subsamples (simple copies) of the vertex-centered values from the children.

TABLE 1
Origin and u_i Axis for Child-to-Parent Mapping

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	$0, 1$	$-\frac{1}{2}, -\frac{1}{2}$

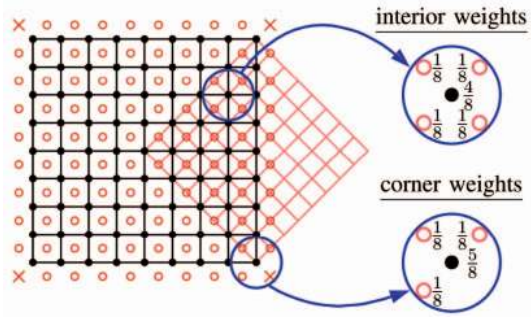

Fig. 8. Low-pass filtering is performed by collecting both cell-centered values (hollow dots) and vertex-centered values (solid dots) from the four children of a diamond. One child is highlighted and the weight masks for the interior and corner cases are given.

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. Section 6 discusses out-of-core processing for coarse-to-fine refinement in general (without ghost cells), while Section 9 covers the special case of fast procedural terrain generation using ghost cells.

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 [21]. 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 Fig. 9. Recall from Knuth [17] that any complete binary tree may be assigned unique indices by setting the root node to 1

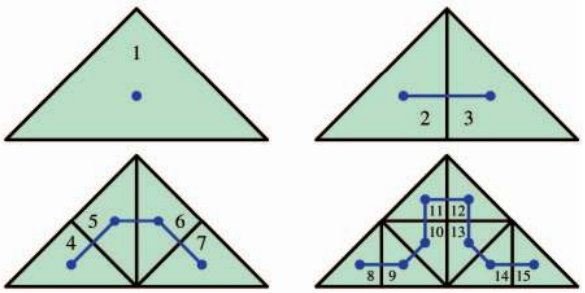

Fig. 9. Sierpinski indices for bintree triangles are computed recursively from their parent index. While the layout is highly coherent, the indices are mapped to triangles, not diamonds.

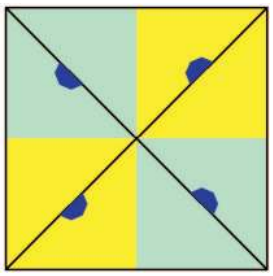

Fig. 10. The 4-edge neighborhood shown is covered exactly once by the diamonds associated with the left edges of the bintree triangles. This pattern repeats to cover the plane. The triangles are shown in outline, the diamond areas in alternating shades, and the diamond centers by marking the inside of their respective triangle's left edge.

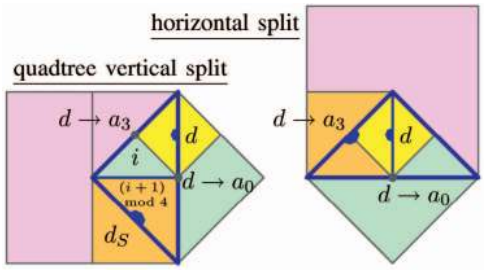

Fig. 11. The Sierpinski parent d_S of a diamond d is determined based on two cases, depending on the orientation of d's quadtree ancestor's distinguished edge. On the left, this edge is vertical and the counterclockwise neighbor of d's left parent is the Sierpinski parent. On the right, the Sierpinski parent is simply d's left parent.

and then, for every node with index k, recursively set its 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 of the midpoints of the shorter edges, say the left side, provides the one-to-one and onto mapping that is needed. Fig. 10 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.

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 Fig. 11, depending on whether the distinguished diagonal of d's quadtree parent $d \to a_0$ is horizontal or vertical. The pseudocode to compute d's Sierpinski index $d \to k$ is then:

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

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:

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

The bits are now of the following form:

$$b_{63}b_{62}b_{61}\ldots b_N 100\ldots 0,$$

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

The "1" mark bit is allowed to be in any of the rightmost four 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 4-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 30 tiles from four 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

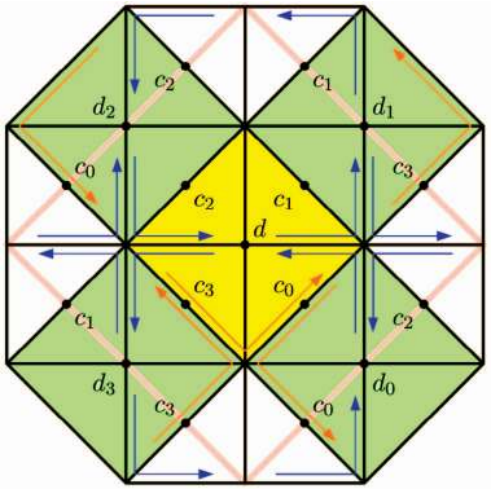

Fig. 12. General fine-to-coarse processing is performed by collecting both cell-centered values and vertex-centered values for the region outlined in pink, processing these according to the operation desired, then storing the results at the vertex-centered locations for the output tile d, shown in yellow. Diamond d's four neighbors $d_{0...3}$ (shown in green) as well as d are used to navigate to the 12 child tiles required. The u, v parameter axes of each tile are shown to explain the mappings in Table 2.

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 Out-of-Core Preprocessing of Planetary Data

Beyond the simple low-pass filtering mechanism given in Section 4, it is important for applications to be able to perform geometry and image-processing operations on tile hierarchies that involve local computations (convolutions, min/max, differences, cross products, etc.). Operations can generally be classified into three types: fine-to-coarse (e.g., low-pass filtering, wavelet analysis), same-level (e.g., shading, image analysis operations like median filtering), and coarse-to-fine (e.g., surface refinement, wavelet reconstruction, procedural terrain generation). All operations are envisioned as reading from an input variable (or variables) and writing to an output variable one tile at a time. Generally, these local operations involve some stencil size, such as a 3×3 neighborhood of elements from the same level for simple median filtering. To perform these operations, the out-of-core process is to:

- **Traverse output tiles in Sierpinski order:** For same-level operations, output tiles may be processed at any level of detail first, or indeed in parallel, since levels do not depend on each other. For fine-to-coarse operations, processing must start with the finest level, then proceed coarser. The order is not strictly fixed, but may be performed on demand for particular subregions of more urgent interest to an application. The gathering operations described below can be used to induce the minimum necessary tile computations for a desired set of output tiles. Similarly, for fine-to-coarse operations, gathering operations can be recursively applied to perform the minimum required work for a desired subset of the possible outputs. In all cases, when multiple output tiles are selected, they are traversed in Sierpinski order to maximize tile-cache coherence.
- **Gather dependent input tiles for the output tile:** The coarse-to-fine, same-level, and fine-to-coarse operations require that certain input tiles in the tile neighborhood be recursively computed first. These dependent tiles are computed one-by-one and copied with appropriate reorientation and translation into a temporary buffer, forming a single raster. These dependencies and mappings involve simple traversals using the diamond data structure.
- **Perform local operations to create output tile:** Once the input tile information has been gathered into a single temporary raster, the requested operation is performed. A wide variety of operations will fit within this framework and can be implemented through call-backs so that applications can extend the set of operations.
- **Maintain a cache of recently used tiles:** All tile access is performed through a fetch system that looks first in cache (keyed by the variable name and Sierpinski index of the tile), then on disk, and, if both of these fail, compute the tile using whatever dependent tiles must be recursively computed. Newly computed tiles are marked so that they will be written to disk when they are about to be recycled from the cache. Overall, any dataflow that can be described as a directed, acyclic graph on variable names and operations can be evaluated through this mechanism.

TABLE 2
Fine-to-Coarse Tile Mappings (Origin and u_i Axis Shown)

child	u_c, v_c	u_a, v_a	child	u_c, v_c	u_a, v_a
$d_0 \rightarrow c_0$	$0, -1$	$\frac{1}{2}, \frac{1}{2}$	$d_2 \rightarrow c_2$	$1, 2$	$-\frac{1}{2}, -\frac{1}{2}$
$d \rightarrow c_0$	$1, 0$	$-\frac{1}{2}, \frac{1}{2}$	$d \rightarrow c_2$	$0, 1$	$\frac{1}{2}, -\frac{1}{2}$
$d_0 \rightarrow c_2$	$1, 0$	$-\frac{1}{2}, -\frac{1}{2}$	$d_2 \rightarrow c_0$	$0, 1$	$\frac{1}{2}, \frac{1}{2}$
$d_1 \rightarrow c_3$	$1, 0$	$\frac{1}{2}, -\frac{1}{2}$	$d_3 \rightarrow c_1$	$0, 1$	$-\frac{1}{2}, \frac{1}{2}$
$d \rightarrow c_1$	$1, 0$	$\frac{1}{2}, \frac{1}{2}$	$d \rightarrow c_3$	$0, 1$	$-\frac{1}{2}, -\frac{1}{2}$
$d_1 \rightarrow c_1$	$2, 1$	$-\frac{1}{2}, \frac{1}{2}$	$d_3 \rightarrow c_3$	$-1, 0$	$\frac{1}{2}, -\frac{1}{2}$

In the previous section, the standard tile parameterization was given for diamonds. These mappings allow the three gathering operations (fine-to-coarse, same-level, and coarse-to-fine) to be implemented with neighbor finding and fixed tables of level-to-level parameter mapping origins and u-axis vectors. For local-operation stencils that do not exceed about half the width of a single tile, only the immediate neighbors of a diamond or its parents are accessed.

Fine-to-coarse tile gathering is depicted in Fig. 12, with the mappings spelled out explicitly in Table 2. The values in this table are the origin and u axis vector of the various

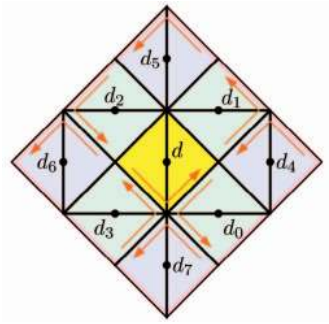

Fig. 13. Same-level processing is performed by collecting vertex-centered values for the region outlined in pink, processing these according to the operation desired, then storing the results at the vertex-centered locations for the output tile d, shown in yellow. Diamond d's four neighbors $d_{0...3}$ (shown in green) as well as four of their neighbors are used as input with the mappings shown.

child diamond coordinate systems with respect to the output diamond d's coordinate system. Twelve tiles at the child level are read for a particular output diamond and are copied to a temporary raster (outlined in pink) for use by the local-operation processing. The mappings in Table 2 must be scaled and biased appropriately for the tile raster sizes used in the application. Stencils that extend up to half the width of the output diamond are supported. This avoids the corner cases that were dealt with in an ad hoc fashion previously in Fig. 8.

The mappings in Table 2, as with Table 1 given earlier, are defined by the origin and one axis vector of a child's parameterization given with respect to output diamond d's parameterization. The neighbors $d_{0...3}$ can be obtained as follows:

$$i_0 \Leftarrow (d \rightarrow i_1 + 1) \bmod 4;\ d_0 \Leftarrow d \rightarrow a_1 \rightarrow k_{i_0}$$
$$i_1 \Leftarrow (d \rightarrow i_1 - 1) \bmod 4;\ d_1 \Leftarrow d \rightarrow a_1 \rightarrow k_{i_1}$$
$$i_2 \Leftarrow (d \rightarrow i_3 + 1) \bmod 4;\ d_2 \Leftarrow d \rightarrow a_3 \rightarrow k_{i_2}$$
$$i_3 \Leftarrow (d \rightarrow i_3 - 1) \bmod 4;\ d_3 \Leftarrow d \rightarrow a_3 \rightarrow k_{i_3}.$$

The various child tiles may be accessed directly from these neighbors, while Table 2 provides the transformation into the output tile d's parameter space so that values may be gathered in the right locations.

A similar mapping is obtained for same-level processing, as shown in Fig. 13. The same-level mappings require accessing neighbors of neighbors. Let $d \rightarrow d_i$ denote the neighbor of d across child i's edge, i.e., the neighbor that has that child in common. The pseudocode given earlier shows how to determine these neighbors. The neighbors $d_{4...7}$ can now be determined as:

$$d_4 \Leftarrow d \rightarrow d_0 \rightarrow d_2$$
$$d_5 \Leftarrow d \rightarrow d_1 \rightarrow d_1$$
$$d_6 \Leftarrow d \rightarrow d_2 \rightarrow d_0$$
$$d_7 \Leftarrow d \rightarrow d_3 \rightarrow d_3$$

or, using modulo arithmetic:

$$d_{4+i} \Leftarrow d \rightarrow d_i \rightarrow d_{(2-i) \bmod 4}.$$

There is a second way to get to each of these neighbors, namely,

$$d_{4+i} \Leftarrow d \rightarrow d_{(i+1) \bmod 4} \rightarrow d_{3-i}.$$

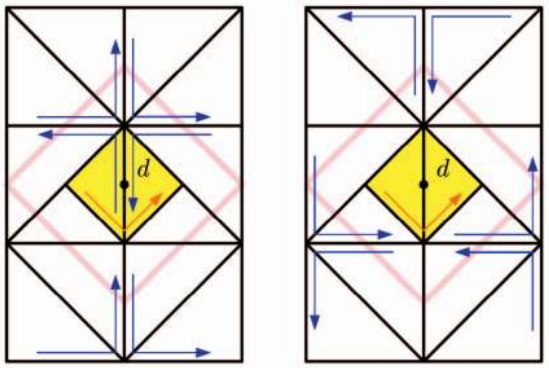

Fig. 14. Coarse-to-fine processing is performed by collecting the cell and vertex-centered values available from the parent-level tiles in the region outlined in pink, processing these according to the operation desired, then storing the results at the vertex-centered locations for the output tile d, shown in yellow. Diamond d's two parents and four of their neightbors are used as input with the mappings shown. The two cases are shown on the left and right.

Typically, this gives exactly the same diamond. However, note that, when one of d's ancestors does not have valence 4 (for example, the cube corners in the planetary base mesh), then the two methods of computing $d_{4...7}$ will yield different answers. Ideally, the filtering operations should be adjusted to handle these cases, but, for simplicity, we just blend between the two possible diamond rasters so as to create a seamless function in these neighborhoods. This situation unfortunately arises at all levels of detail around non-valence-four base-mesh vertices.

There are six parent-level tiles that must be gathered to form an input raster for general coarse-to-fine processing, as depicted in Fig. 14.

6.1 Out-of-Core Preprocessing Example: Lake Detection

Lake detection can be performed automatically using image-processing operations in sequence. For the Washington-state 10 meter data set, we use a flatness detector to indicate all potential lake cells, followed by several image erosion steps (eliminating lake cells where not all of the eight neighbors are lake cells), and ending with a large number of image dilation steps (adding back flat-neighborhood cells that are adjacent to the current lake cells). This processing was performed on the complete data set in well under an hour, including shading and hierarchy building, using only a small memory footprint of less than 100 megabytes for this 2.4 gigabyte data set. Results of the detection and shading process are shown in Fig. 15.

7 Geometry Patches and Frame-to-Frame Updates

When replacing individual leaf triangles with small patches of, say, 1,024 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.

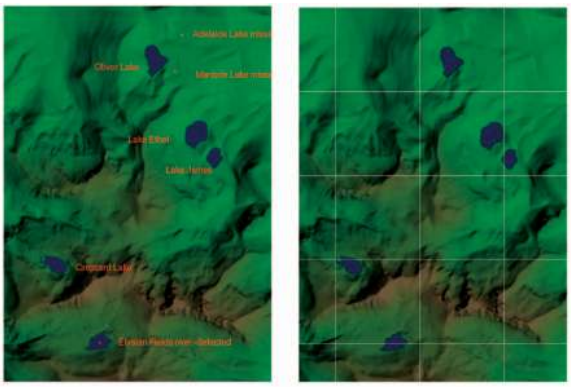

Fig. 15. Detail from the lake detection processing results for the 10 meter Washington state data set, just north of the peak of Mount Rainier. Small lakes are a challenge to the noise reduction scheme employed so that two lakes in the upper right were missed (Adelaide and Marjorie) due to the number of erosion steps. Lakes Oliver, Ethel James, and Crescent were detected properly. Elysian Fields was detected as a much larger body of water than it actually is, probably because that area is flat. The second image outlines the fine-resolution raster tiles on which the operations were performed, demonstrating that tile-based processing does not create boundary artifacts.

From [26], we know that, for any uniform refinement of a right isosceles triangle that is a power of four, such as 256 or 1,024, 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 Fig. 16 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.

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 trade-off to be a tile with 129 or 257 vertices (elevation samples) per side. For triangular patches, either 256 or 1,024 triangles are used. Fig. 16 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 four outer edges of the patch diamonds and allow their interiors to be smoothed out through low-pass filtering. For example, in Fig. 16, 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 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

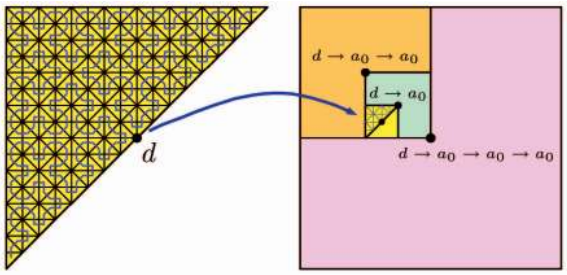

Fig. 16. The Sierpinski layout of a triangle patch, with the mapping of the patch to its elevation tile. If d is the diamond of the triangle patch, then the child-to-parent mappings of Section 4 can be composed to locate the appropriate elevation values in the third quadtree ancestor, $d \to a_0 \to a_0 \to a_0$.

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.

Similarly to ROAM, dual-queues are used to prioritize, respectively, diamond split and merge activity. Unlike the ROAM base priority, which 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.

8 4-8 Textures for Triangle Patches

Most multiresolution 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 triangle 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 using a single priority queue, so as to keep α close to 1.0. Higher priority is given to coarsening a patches' texture association as α becomes greater than one and refining becomes a higher priority as α becomes less than one. This is summarized by defining the update priority as

$$p(\alpha) = \begin{cases} \|\alpha - \sqrt{2}\| & \text{if } \alpha \geq 1 \\ \|1/\alpha - \sqrt{2}\| & \text{if } \alpha < 1. \end{cases}$$

No updates should be taken for priorities $p(\alpha) \leq 0$ since this is the threshold at which the update would result in α farther from 1.0 than the current mapping. We keep to a budget of four to eight 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 texture objects are loaded, we keep coarser textures so that the system can always instantly coarsen as desired. The texture-object diamonds are optimized frame-to-frame using a similar dual-queue optimization loop that was used for triangle patches, using $p(\alpha)$ as the split/merge priority, where α is computed for an entire texture object as a conservative estimate of the worst-case patch that might map to it. As with the dual-queue triangle-patch optimization, tile access and texture upload operations are scheduled for a subsequent load phase and are limited by a per-frame budget.

In the case of pure height maps (as opposed to full planetary geometry), it is possible to use automatic texture coordinate generation (OpenGL's glTexGen call or texture coordinate transforms, for example). Our experience with implementations indicates that explicit texture coordinates per vertex have higher performance, at some increase in vertex-memory use. This performance penalty is due to the expensive state changes that must be made per texture object, given that textures are broken up into many tiles with independent coordinate systems. Also, actual hardware and drivers generally run slower when nonidentity texture coordinate transforms or glTexGen is used. Finally, the use of these automatic texture coordinate schemes is not applicable to planetary geometry or other nonplaner base meshes.

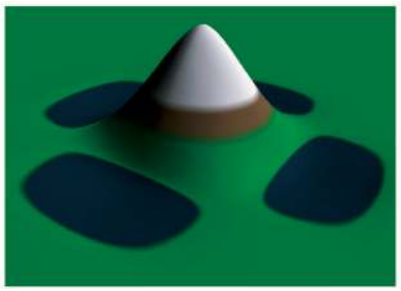

Fig. 17. This figure depicts the relative effect of one random displacement in the procedural terrain generation. It is the smooth interpolatory basis resulting from 4-8 refinement, pseudocolored to highlight the negative lobes in the displacement field as blue. The quality of this basis, namely, its symmetry, smoothness, and lack of excessive oscillations, are fundamental to the overall perceived (and statistically measured) qualities of a procedural terrain.

9 Procedural Terrain Refinement

The basic one-dimensional interpolatory refinement used is the following: Let f_i be values before refinement, where i is an integer. There will be twice as many refined values g_j defined by copying the old values verbatim and using weighted averaging to get new midpoints:

$$g_{2i} = f_i$$
$$g_{2i+1} = -\tfrac{1}{16} f_{i-1} + \tfrac{9}{16} f_i + \tfrac{9}{16} f_{i+1} - \tfrac{1}{16} f_{i+2}.$$

To perform two-dimensional refinement, one can conceptually apply the one-dimensional refinement along each row in a raste, and then along each resulting column. For 4-8 refinement, the midpoints are added every odd level and the edge points are then added the next even level. After two steps, this exactly reproduces the output of the computation that works on one axis at a time. The smooth basis function that results is shown in Fig. 17.

To speed up the refinement processing in this case, it is possible to avoid the general gather operation described in Section 6 by adding ghost values to each tile (raster elements that are duplicated a little ways into the neighbor tile and maintain the same values as their duplicate elements). It turns out to be impossible to maintain sufficient neighbors if a tile gathers from its two parents, but only two ghost values must be added beyond the common tile edge values if the tile information is gathered from the next-coarser even-level tile, due to the two-stage subdivision process.

Additional geometric detail is added using a smooth interpolatory subdivision scheme on tiles, as just described, combined with random displacements that grow smaller exponentially as the level-of-detail becomes finer. Random displacements are only added at even-level tiles, consistent with the implementation of the interpolatory refinement with ghost values (note that the ghost values should be displaced the same as their duplicates in the neighbor tile). An example of terrain generated completely using these procedural constructions is shown in Fig. 18. The use of procedural detail to aid visualization of actual terrain data (a Mars polar region) is shown in Fig. 19.

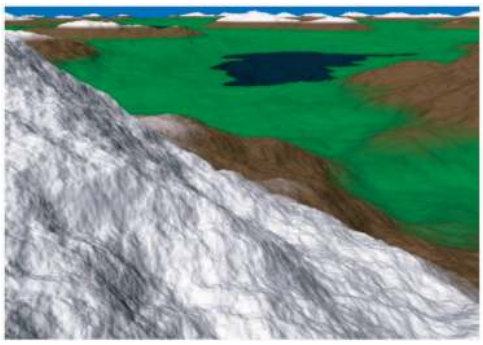

Fig. 18. A screen shot during navigation through a completely procedural (synthetic) landscape, used to demonstrate the feasibility of the proposed method for more than just adding detail to actual elevation databases. A fast, smooth, interpolatory subdivision method was used, along with random displacements produced so that the terrain is fully deterministic (the same frustum will produce the same view regardless of the navigation history). This screenshot (actual size 739×541 pixels) was taken on a Linux PC with a 2.53GHz Pentium 4 with RDRAM 1066 memory and an Nvidia GF4 Ti4600 graphics card. This view displays 1.56 million triangles at 25.6 frames per second, totaling 40 million triangles per second rendering rate. This is slightly under the maximum possible rate for this particular hardware, view frustum, screensize, and collection of textured triangles. This reduction is due to use of small 256-triangle patches to achieve higher adaptivity.

10 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 [27] 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.

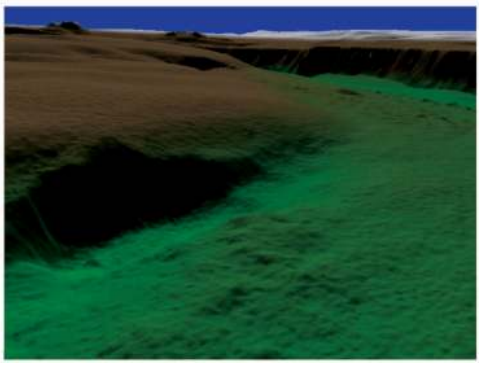

Fig. 19. Procedural detail is added here to a Mars polar region to aid in the visualization of this data set by adding visual cues to what would otherwise be an unrealistically smooth surface that would be hard to discern. The elevations in this case are highly exaggerated, as requested by the planetary geologists examining this data.

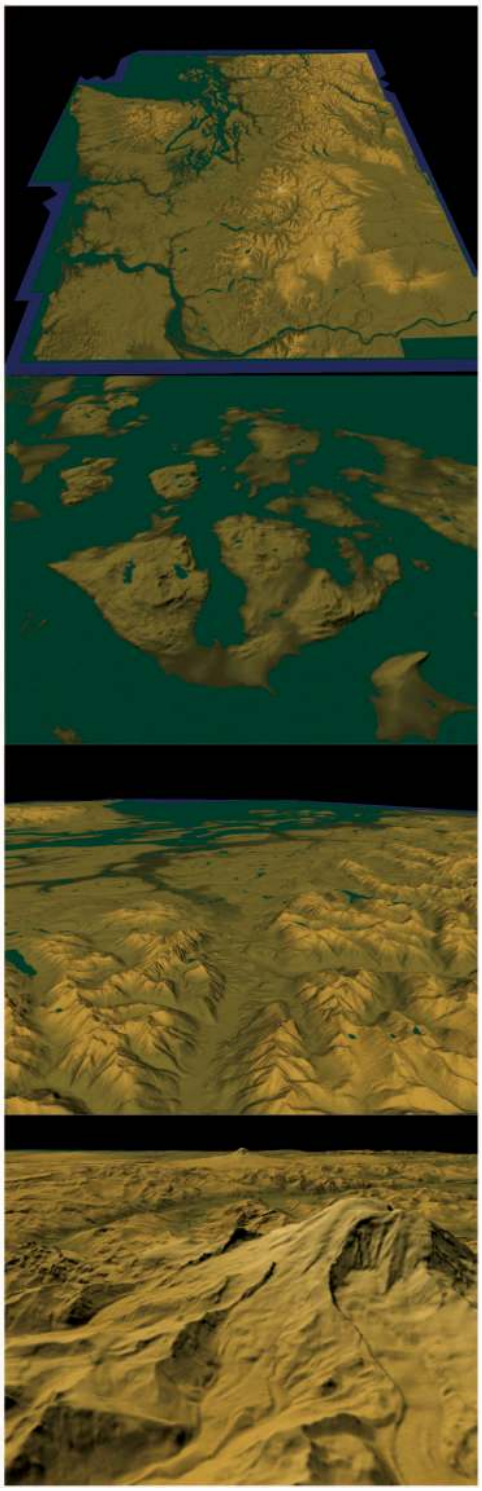

Fig. 20. Screen shots of our test flight showing the overall Washington state data set, the San Juan islands, a view facing Victoria, and Mount Rainier with Mount Adams behind.

In the rendering application, approximately 53 percent 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 percent 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 loops is a user-defined parameter, but, in this test, less than 2 percent time was spent on this. Less than 1 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. 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 our previous paper [16], including detailed performance graphs over a test flight. Snapshots from the flyover are highlighted in Fig. 20.

11 Conclusion

The paper has studied the use of a new diamond data structure to represent view-dependent adaptations of 4-8 meshes. Tiles are used per diamond for both geometry and texture and show very high quality antialiasing through precise low-pass filtering, given that diamond hierarchies have twice as gradual stepping to lower-frequency representations as conventional quadtree-like schemes. Sierpinski out-of-core indexing is introduced and was shown to facilitate massive-data preprocessing as well as runtime paging during frame-to-frame view-dependent optimization. A general framework for batch preprocessing of raster hierarchies was presented that utilizes coarse-to-fine, same-level, and fine-to-coarse tile gather operations. Preprocessing was shown to be fast on massive data sets using only a small, fixed memory footprint. A simple but visually pleasing procedural terrain generation method was described and shown to be very fast to compute during a real-time fly-though.

Future work based on this terrain system can be expanded to include dual queues at all levels of cache, for both geometry and texture. This would replace the reactive, least-recently-used strategy with a system that supports 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 real-time, high quality procedural detail is also of interest, beyond the simple random displacement scheme used here.

Acknowledgments

This work was performed under the auspices of the US Department of Energy by the University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. This work was supported by the US 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 (LLNL) under contract B523818. The authors thank the members of the Visualization and Graphics Group of the Institute for Data Analysis and Visualization (IDAV) at the University of California, Davis, and the visualization researchers within the Center for Applied Scientific Computing at LLNL. They especially thank the anonymous reviewers, whose thoughtful suggestions for improvements to the paper have been of great value.

References

[1] Nat'l Aeronautical and Space Administration, "MOLA Data Set," http://pds-geosciences.wustl.edu/missions/mgs/megdr.html, 2003.

[2] L. Balmelli, J. Kovacevic, and M. Vetterli, "Quadtrees for Embedded Surface Visualization: Constraints and Efficient Data Structures," *Proc. IEEE Int'l Conf. Image Processing (ICIP),* pp. 487-491, Oct. 1999.

[3] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "BDAM: Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization," *Proc. Eurographics 2003,* pp. 505-514, Sept. 2003.

[4] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Interactive Out-of-Core Visualization of Very Large Landscapes on Commodity Graphics Platforms," *Proc. Int'l Computer Vision Symp. (ICVS 2003),* pp. 21-29, Nov. 2003.

[5] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM)," *Proc. IEEE Visualization,* pp. 147-155, Oct. 2003.

[6] C. DeCoro and R. Pajarola, "XFastMesh: Fast View-Dependent Meshing from External Memory," *Proc. IEEE Visualization,* pp. 363-370, 2002.

[7] J. Döllner, K. Baumann, and K. Hinrichs, "Texturing Techniques for Terrain Visualization," *Proc. IEEE Visualization,* pp. 227-234, 2000.

[8] M.A. Duchaineau, M. Wolinshy, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineev-Weinstein, "ROAMing Terrain: Real-Time Optimally Adapting Meshes," *Proc. IEEE Visualization,* pp. 81-88, Oct. 1997.

[9] J. El-Sana and Y.J. Chiang, "External Memory View-Dependent Simplification," *Computer Graphics Forum,* vol. 19, no. 3, pp. 139-150, Aug. 2000.

[10] W. Evans, D. Kirkpatrick, and G. Townsend, "Right Triangular Irregular Networks," Technical Report TR97-09, Dept. of Computer Science, Univ. of Arizona, May 1997.

[11] W. Evans, D. Kirkpatrick, and G. Townsend, "Right-Triangulated Irregular Networks," *Algorithmica,* vol. 30, 2001.

[12] L. De Floriani and E. Puppo, "Hierarchical Triangulation for Multiresolution Surface Description," *ACM Trans. Graphics,* vol. 14, no. 4, pp. 363-411, 1995.

[13] R.J. Fowler and J.J. Little, "Automatic Extraction of Irregular Network Digital Terrain Models," *Computer Graphics (SIGGRAPH '79 Proc.),* vol. 13, no. 3, pp. 199-207, Aug. 1979.

[14] T. Gerstner, "Multiresolution Compression and Visualization of Global Topographic Data," *GeoInformatica,* vol. 7, no. 1, pp. 7-32, 2003.

[15] H. Hoppe, "Smooth View-Dependent Level-of-Detail Control and Its Application to Terrain Rendering," *Proc. IEEE Visualization,* pp. 35-42, Oct. 1998.

[16] L.M. Hwa, M.A. Duchaineau, and K.I. Joy, "Adaptive 4-8 Texture Hierarchies," *Proc. IEEE Visualization,* pp. 219-226, Oct. 2004.

[17] D.E. Knuth, *The Art of Computer Programming, Sorting and Searching,* second ed., 1975.

[18] J. Levenberg, "Fast View-Dependent Level-of-Detail Rendering Using Cached Geometry," *Proc. IEEE Visualization,* pp. 259-266, Oct. 2002.

[19] P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hughes, N. Faust, and G. Turner, "Real-Time, Continuous Level of Detail Rendering of Height Fields," *SIGGRAPH 96 Conf. Proc.,* pp. 109-118, Aug. 1996.

[20] P. Lindstrom and V. Pascucci, "Visualization of Large Terrains Made Easy," *Proc. IEEE Visualization,* pp. 363-370, 2001.

[21] P. Lindstrom and V. Pascucci, "Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization," *IEEE Trans. Visualization and Computer Graphics,* vol. 8, no. 3, pp. 239-254, July-Sept. 2002.
[22] F. Losasso and H. Hoppe, "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids," *SIGGRAPH 2004 Conf. Proc.,* pp. 769-786, Aug. 2004.
[23] A. Mirante and N. Weingarten, "The Radial Sweep Algorithm for Constructing Triangulated Irregular Networks," *IEEE Computer Graphics and Applications,* vol. 2, no. 3, pp. 11-13, 15-21, May 1982.
[24] T. Möller and E. Haines, *Real-Time Rendering,* second ed. A.K. Peters Limited, 1999.
[25] R. Pajarola, "Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation," *Proc. IEEE Visualization '98,* pp. 19-26, 515, 1998.
[26] A. Pomeranz, "ROAM Using Surface Triangle Clusters (RUSTiC)," MS thesis, Dept. of Computer Science, Univ. of California, Davis, June 2000.
[27] US Geological Service, "State of Washington Data Set," http://rocky.ess.washington.edu/data/raster/tenmeter/onebytwo10/index.html, Dec. 2001.
[28] C.T. Silva, J.S.B. Mitchell, and A.E. Kaufman, "Automatic Generation of Triangular Irregular Networks Using Greedy Cuts," *Proc. IEEE Visualization,* pp. 201-208, 1995.
[29] C.C. Tanner, C.J. Migdal, and M.T. Jones, "The Clipmap: A Virtual Mipmap," *SIGGRAPH '98 Conf. Proc.,* pp. 151-158, July 1998.
[30] T. Ulrich, "Rendering Massive Terrains Using Chunked Level of Detail Control," *SIGGRAPH Course Notes,* 2002.
[31] L. Velho, "Using Semi-Regular 4-8 Meshes for Subdivision Surfaces," *J. Graphics Tools,* vol. 5, no. 3, pp. 35-47, 2000.
[32] L. Williams, "Pyramidal Parametrics," *SIGGRAPH '83 Proc.,* vol. 17, no. 3, pp. 1-11, July 1983.

Lok M. Hwa recently received the MS degree from the Department of Computer Science at the University of California, Davis, within the Institute for Data Analysis and Visualization (IDAV). His interests include real-time display of large texture and geometry databases, film-making, visual effects, and realistic rendering. He is pursuing a career in the film industry.

Mark A. Duchaineau received the PhD degree in computer science from the University of California, Davis, in 1996 and joined Lawrence Livermore National Laboratory (LLNL) the same year. He is the project leader for scientific visualization within the Center for Applied Scientific Computing (CASC). He is the author of numerous research papers in the areas of scientific visualization and multiresolution methods, among which are included wavelet analysis of large semistructured grids, real-time optimization of grids for display on graphics hardware, large-scale compression for visualization, and hierarchical splines with output sensitive approximation. He is active in reviewing papers for several journals and conferences, including IEEE Visualization (program committee), ACM SIGGraph, and *IEEE Transactions on Visualization and Computer Graphics*, and is a member of the IEEE, ACM, and SIAM.

Kenneth I. Joy is a professor of computer science at the University of California at Davis, where he is codirector of the Institute for Data Analysis and Visualization (IDAV). He joined UC Davis in 1980 in the Department of Mathematics and was a founding member of the Computer Science Department in 1983. He is a faculty computer scientist at the Lawrence Berkeley National Laboratory and a participating guest researcher at the Lawrence Livermore National Laboratory. His primary research interests are in the areas of visualization, multiresolution representation of data, geometric modeling, and computer graphics. He serves on the editorial board of the *IEEE Transactions on Visualization and Computer Graphics* and served as a papers cochair and proceedings coeditor for the IEEE Visualization conferences in 2001 and 2002. He is a member of the IEEE and the IEEE Computer Society.