Adaptive 4-8 Texture Hierarchies

Lok M. Hwa[*] Mark A. Duchaineau[†] Kenneth I. Joy[*]

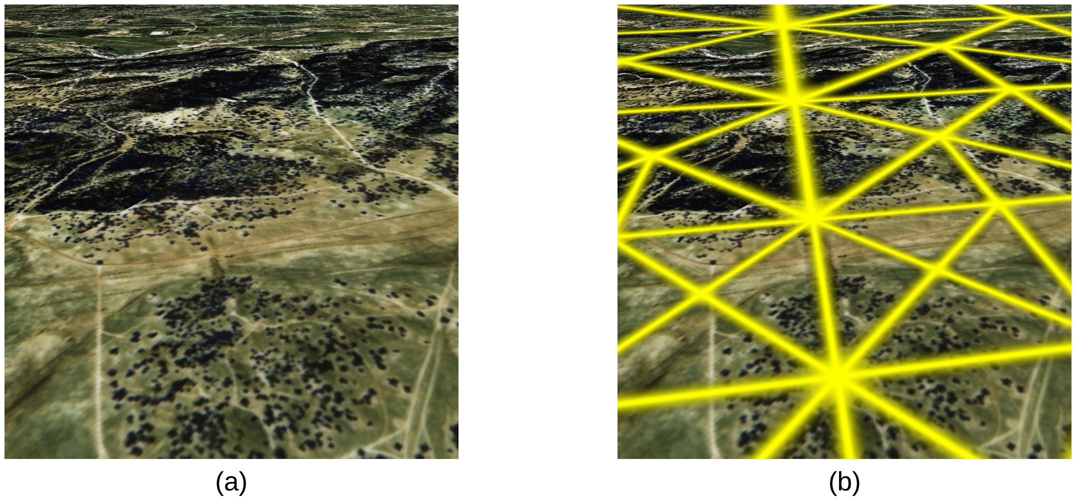

Figure 1: Two screen shots of an overflight of Fort Hunter Liggett, CA that illustrate the use of 4-8 texture hierarchies. Figure 1a shows the seamless textured image produced by the system, while Figure 1b shows the outline of the texture tiles used in producing the image.

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.

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 [17]. Dynamic adaptation 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 100 million triangles per second, this means that choosing triangle adaptations for uniform screen size will result in roughly one-pixel triangles for modest window sizes 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.[1] 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 ef-

[*]Center for Image Processing and Integrated Computing (CIPIC), Department of Computer Science, University of California, Davis, CA, {lmhwa,kijoy}@ucdavis.edu

[†]Center for Applied Scientific Computing (CASC) Lawrence Livermore National Laboratory, Livermore, CA, duchaineau1@llnl.gov

[1]Graphics hardware uses point sampling per pixel when rasterizing triangles.

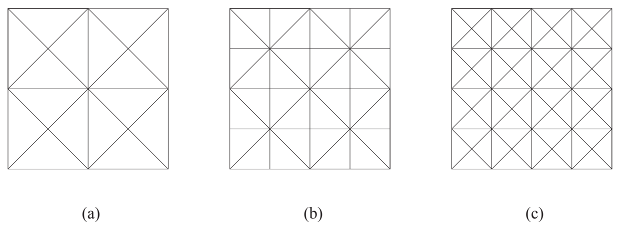

Figure 2: A 4-8 mesh illustrating different levels of mesh resolution. Note that each vertex has a valence of either four or eight.

fectively 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]. We chose the 4-8 meshes since these match the constraints of texture hardware and have many known desirable properties [12, 6, 13], see Figure 2.

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 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 use for 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 [21, 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 6.

For geometric rendering, *chunks* 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 [16, 11]. We are able to achieve triangle throughput close to the practical limits on recent PC video cards. Section 4 outlines how chunks are laid out and updated.

The adaptive 4-8 textures, defined in detail in Section 5, 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 chunk 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 chunk's parameterization to the texture diamond's parameter space is computed as needed when this level-of-detail selection changes. This change requires and 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 operations. Therefore these operations 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

Many multiresolution geometry schemes exist for terrain rendering; a complete review of these systems is beyond the scope of this paper. Two primary systems are utilized to store and simplify the terrain geometry. Systems that use a regular grid approach are built on a hierarchy of right triangles [14, 7], while triangulated irregular networks (TINs) [8, 18, 9] work to solve this problem by not restricting triangulations to a regular grid. Both schemes have advantages and disadvantages. We utilize regular grids for efficiency and uniformity of treatment and we review the related work in this area.

Several researchers have based terrain rendering on regular grids. Lindstrom *et al.* [12] present a method based upon an adaptive 4-8 mesh, using longest-edge bisection (LEB) as a fundamental operation to refine the mesh. They use an elegant bottom-up vertex-reduction method to reduce the size of the mesh for display purposes. Duchaineau *et al.* [6] present a system for visualizing terrain also based upon a LEB paradigm. Their system uses a dual-queue management system that splits and merges cells in the hierarchy according to the visual fidelity of the desired image. Lindstrom and Pascucci [13] describe a framework for out-of-core rendering and management of massive terrain surfaces. They present a view-dependent refinement method along with a scheme for organizing the terrain data to improve coherence and reduce the number of paging events from external storage to main memory. Again, they organize the mesh using a longest-edge bisection strategy, using triangle stripping, view frustrum culling and smooth blending of geometry. Pajarola [15] utilizes a restricted quadtree triangulation, similar to an adaptive 4-8 mesh for terrain visualization.

Large texture processing has been attempted by several researchers. One of the first methods of prefiltering texture levels of detail was described by Williams' [22]. His "mipmaps" were defined to be a collection of images of increasingly reduced resolution, arranged loosely as a pyramid. Starting with level zero, the largest and finest level, each lower level represents the image using half the number of texels in each dimension. For the two-dimensional case, this implies that level k contains one-quarter of the texels of level $k-1$. Per-pixel rendering with a mipmap is accomplished by projecting the pixels into mipmap space using texture coordinates and transformations to define the projection. Each rendered pixel is derived from one or more texel samples taken from one of more levels of the mipmap hierarchy. The mipmap hierarchy is generated by prefiltering texture levels.

Tanner *et al.* [19] expanded upon the idea of mipmaps, introducing clipmaps. This method also utilized levels of an in-core texture pyramid with levels of resolution that differed by a factor of four, but allowed for arbitrarily large textures. This algorithm utilized the fact that the complete mipmap pyramid is rarely used during the rendering of a single image and much of the pyramid could be clipped away allowing much larger textures to be used.

Ulrich [20] presented a method of texture "chunking" to handle large out-of-core meshes. This method utilizes a tree of static preprocessed meshes. The tree and its component meshes, the "chunks" are generated in a preprocessing step. Each chunk is a static mesh that can be rendered with a single draw primitive. The chunk at the root of the tree is a low-detail representation of the entire object. The child chunks of the root node split the object into several pieces, and each piece independently represents its portion of the object with a higher level of detail than the parent. The tree is organized as a quadtree of height field sub-squares. Upon rendering, the chunks are chosen to meet the desired visual fidelity.

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 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. Hierarchical view frustrum culling and view-dependent texture and geometry refinement are both performed each frame. Textures are managed as rectangular tiles, resulting in an quadtree representation of the textures. The rendering system traverses the texture quadtree until acceptable error conditions are met, and then selects corresponding patches in the geometry bintree system. Once the texture has been chosen, the geometry is refined until the geometry space error is within tolerance.

The difficulty with adaptively texturing a dynamic mesh is creating the illusion of a static mesh with static textures, given limited computing resources. Swapping between textures of various resolutions for a neighboring triangles must be imperceptible and seams must be invisible. The focus of our research is to address these issues in the real-time rendering environment for out-of-core data sets. We introduce additional texture levels of detail by using an adaptive 4-8 mesh structure coupled with a diamond backbone data structure and queues for processing both geometry and texture. Our method can process massive textures in a multiresolution out-of-core fashion for interactive rendering of textured terrain.

3 The Diamond Data Structure

A 4-8 refinement of the plane starts with a square grid, adds the centroids of each square cell, and rotates the square grid 45 degrees and rescales by $\sqrt{\frac{1}{2}}$ to obtain another square grid at the next finer level of resolution, as shown in Figure 3. The edges of the previous level's grid can be retained as distinguished diagonals. In this setting, we define a *diamond* to be a square at a given level of resolution, along with its center point and distinguished diagonal. Each vertex, edge or cell of the 4-8 hierarchy maps to exactly one diamond. Thus a diamond becomes the single element needed to represent the elements of 4-8 adaptive meshes, and forms the backbone for applications involving geometry or texture tiles.

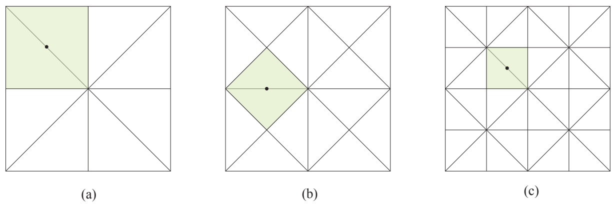

Figure 3: Diamonds at various levels of resolution in a 4-8 mesh. As the resolution increases, the diamonds are rotated 45° and scaled by $\frac{\sqrt{2}}{2}$.

For incremental updates to an adaptive refinement, preprocessing operations, and other general mesh operations, it is necessary to support traversals of a diamond-based mesh. Each diamond has two parent diamonds, several neighboring diamonds, and four children. A diamond is represented by eight pointers: two pointers to the diamond's two parents, two pointers to ancestor diamonds, and four pointers to child diamonds.[2] By using a standard orientation in labeling these pointers, the various relationships needed for the traversals become readily apparent, as shown in Figure 3. This standard orientation involves listing the ancestors and children in counter-clockwise order, and anchoring the diamond's parameterization at the unique corner which formed by *quad parent*, a vertex of valence four in the 4-8 mesh. We label the quadnode q, the two parent diamonds p_1 and p_2 and the fourth parent a. The children are denoted c_0, c_1, c_2, c_3 in counterclockwise order. p_1 is the right parent, p_2 is the left parent and a is an older ancestor in the refinement. The children are indexed so that child c_i is on the edge between parent and ancestor. It is a straightforward pointer manipulation problem to generate and link the children to the parents.

Each diamond is the child of two parents. Traversals to parents and children are matters of following direct links. Edge-neighbors of diamond d are siblings of either its left or right parent, either clockwise or counterclockwise from d. To move to a neighbor it is therefore convenient to keep in d its child index with respect to both its left and right parents.

Conformant (crack-free) triangulations are insured by the requirement that both parents of a diamond must be created before the diamond can be created.

We utilize the dual split/merge queue paradigm of Duchaineau *et al.* [6] to manage the diamond structure. When a diamond comes into existence it introduces the shared base-edge of its two right triangles. If d is a diamond on the split queue, the split operation insures that the four children of d are created.[3] The split diamond is then placed on the merge queue and the newly created children on the split queue.

Split and merge priorities are computed using the projected maximum distortion for a given diamond, similar to the original ROAM

[2] After extensive performance profiling of several alternatives, including a very memory-lean scheme using (i, j) index arithmetic, space-filling curves, and self-optimized hash lookups, it turns out to be far faster to use a purely pointer-based scheme to traverse to parents, neighbors and children.

[3] A diamond has two parents and either parent can create it in the split operation. When splitting, it may not be necessary to create all four children of a diamond, as some may have been created by a previous split.

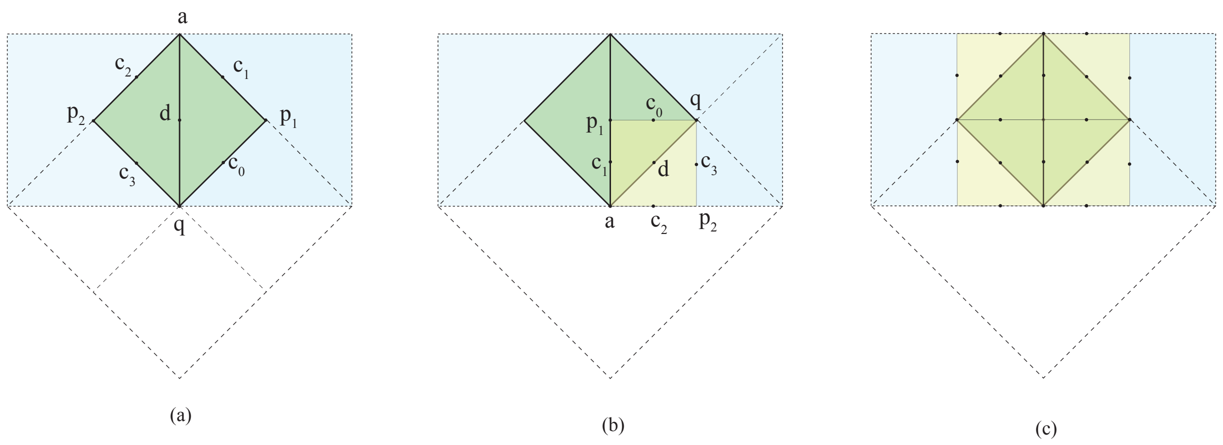

Figure 4: The standard index scheme for a diamond (green) diamond is shown in (a). d is the diamond index and uniquely specifies each diamond. The left and right parents are denoted p_1 and p_2, respectively. The quad parent is q and the ancestor parent is a. The indexing scheme for one child diamond (yellow) is shown in (b), and the pointer correspondences between parent and child pointers can be easily seen. The indexing scheme imposes a counterclockwise orientation of diamonds in the mesh. All four children (yellow) of the green diamond are shown in (a).

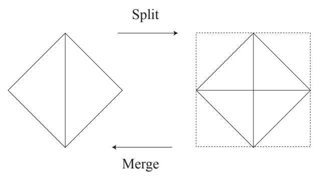

Figure 5: The split and merge operations. The split operation creates the four children of a diamond, allowing the four triangles inside the original diamond to be used in the geometry or texture. The merge operation reverses the split.

method [6]. These priorities are then mapped to indices in bucket queues for either splits or merges. The split queue is prioritized by maximum error so that we choose to split areas of the mesh that display the most visual error in the scene. Prioritizing the merge queue by minimum error reduces detail in areas that display minimal screen distortion. In cases where the priority queues become considerably large in size, priority computations can be deferred to guarantee a time bound. A simple and effective approach is to limit the number of priority updates per frame to a fixed size.

4 Data Tiles and Triangle Chunks

Geometric adaptation for a given viewpoint must be performed as fast as the rendering of geometry to be most useful. Older view-dependent methods did CPU work per frame on every triangle to be output. More recent work has focused on performing CPU work on coarser-grained units, herein called *chunks*, which for greatest efficiency should be static for several frames, and should also be carefully laid out in vertex arrays so as to make maximum use of the vertex caches on the graphics card. In this work, we replace each diamond triangle with a chunk of around 256 or 1024 triangles, taken as though the diamond's triangles were uniformly refined. Any two additional levels of refinement will end up splitting each chunk edge into twice as many segments, and thus will create chunks that adapt seamlessly during selective refinement of the 4-8 mesh.

Since our goal is to provide nearly uniform pixel-sized triangles, we do not need to evaluate detailed error metrics per triangle within a chunk. Instead, for our purposes it is most helpful to create a sphere bound with radius $r(d)$ and center (split) vertex $v(d)$ that just encloses all the triangles within the two chunks that make up a diamond d. We then estimate the number of pixels per triangle using a simple projection of this sphere into screen space. The formula for a chunked diamond's priority is:

$$p(d) = \frac{(f_{\text{frust}}\, r(d))^2}{K\left\|v_{\text{eye}} - v(d)\right\|_2^2}$$

where K is the number of triangles total in a diamond's two chunks, v_{eye} is the current camera position, and f_{frust} is a factor related to the current window size and angle of view that correctly scales an epsilon-radius sphere into pixel radius at the center of the window. A priority value $p(d) = 1$ represents an estimate that each triangle projects to one pixel. Priorities larger than a preset value, say $p(d) > \sqrt{2}$, a split operation is desirable, whereas for $p(d) < \sqrt{1/2}$, a merge is preferable. If the total triangle budget is low for a frame, then triangles greater than $\sqrt{2}$ pixel area will be used, and the dual-queue split-merge processing will naturally organize even projected areas.

For efficiency, chunks are created on the fly during split-merge operations from a more compact raster of normal-displacement values, which we call *tiles*. In our implementations either tiles are either 129 squared or 257 squared rasters. During preprocessing the tiles are traversed recursively, where each diamond tile requests that each of its four children create their low-pass tiles, and then gathers these together and performs its own low-pass filtering. The tile grids of the four children provide both vertex-center values covering the diamond's raster, as well as cell-centered values, as shown in Figure 7. The basic low-pass filtering kernel we use is to weight the vertex value by $1/2$, and the four neighboring cell values by $1/8$, as shown in the Figure.

Low-pass filtering, rather than the simple subsampling used in older view-dependent algorithms, is now both desirable and possible when using chunks. To maintain a crack-free mesh, only the boundaries of chunks must agree. We low-pass filter the interior of a diamond using the data from its children. The boundary and corner vertices are subsampled. We observe that these boundaries

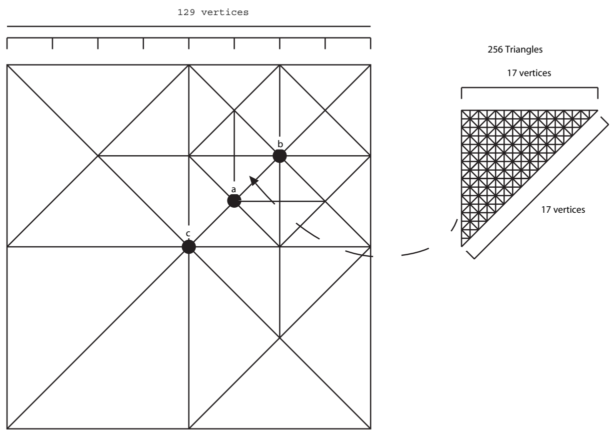

Figure 6: Each of the two triangles of a diamond is associated with 256 triangles from a 129x129 grid of triangles. To select the correct tile for a diamond d, we use the third quad parent of d, which is a diamond whose tile contains the correct 129x129 grid.

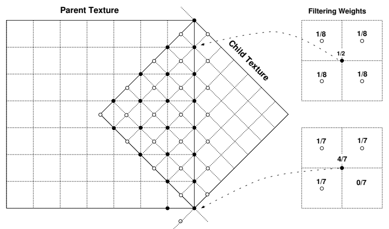

Figure 7: Low-pass filtering from fine (child) to course (parent) levels of resolution.

become interiors at the next coarser level of resolution, and thus have the opportunity for higher-quality filtering at that point. Ultimately only the vertices that persist all the way to the base mesh never get low-pass filtered.

Vertex and index arrays for a chunk are laid out in Sierpinski order, which in our tests provides excellent performance on current PC graphics cards.

5 Adaptive 4-8 Textures

Most multi-resolution texture algorithms use a prefiltered quad-tree of textures. Textures are resampled such that each lower level of detail is one-fourth the area of the previous level creating a mipmap pyramid. Selecting levels of resolution that differ by factors of four produces visual discontinuities in a mesh composed of different texture detail levels, when applied on a per-polygon or per-triangle-chunk basis. Our method creates twice as many detail levels, allowing a smoother transition between levels, while effectively using the diamond hierarchy for level traversal.

The initial data set texture is diced into 128 x 128 tiles, which represent the texture at the finest level. A parent diamonds texture is taken from a weighted average of its four child textures. The filtering approach from level-to-level preserves the energy of the original signal in the texture and filters out more high-frequency detail with each pass. Geometry filtering is performed similarly with the edge values of the parent diamonds being copied over from the child level to maintain continuity when triangles neighbor each other at one level apart.

Each triangle is evaluated to determine the “appropriate” texture resolution that should be applied. The highest resolution texture is six triangle levels below the geometry level. This relationship is maintained by a diamond always displaying the geometry from three quad parent diamonds up in the hierarchy. After a level of detail is chosen, a lower resolution texture is found by traversing up the diamond hierarchy from child to parent. Level of detail calculations are based on the texel-to-pixel ratio per triangle. Using the bounding sphere radius previously calculated for frustum culling, we compute an upper bound on the possible screen area covered by the diamond data. The maximum screen space coverage occurs when looking at a diamond “head-on.” If the desired texture is not cached in texture memory, a request is made by adding the corresponding diamond to a texture-wait bucket queue with priorities defined as the number of jumps up the hierarchy. Priority is given to requests with fewer jumps since this is associated with a higher level of detail and a closer distance to the eye point. The next available texture in the diamond child-parent hierarchy is then applied to the triangle. Because updates to texture memory are expensive, the wait queue allows a fixed number of textures to be uploaded per frame, also avoiding irregular load times.

6 Out-of-core Processing

For efficient input and output, files and disk blocks are laid out using a diamond indexing scheme based on the Sierpinski space-filling curve.

The Sierpinski curve arises from a simple indexing scheme for 4-8 meshes. Each triangle of the root diamond can be split into two triangles. The 4-8 scheme effectively splits each of these triangles into a triangle bintree. The nodes of triangle bintrees can be ordered by assigning the value 1 to the root node, and the values $2k$ and $2k+1$ to the children of node k, see [10]. These values are called the

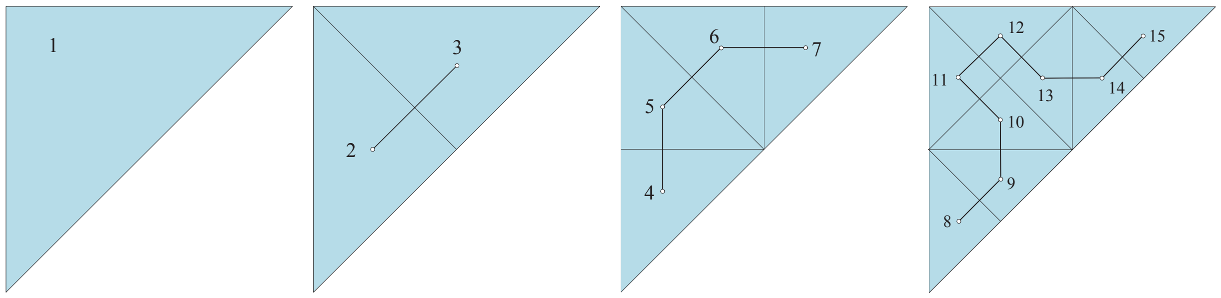

Figure 8: Sierpinski Indices: Note the index ordering implies the Sierpinski curve.

Sierpinski indices of a node, as the enumeration of the nodes of the bintree traces out the Sierpinski curve, see Figure 6. The Sierpinski indices can be used to establish coherent layouts of tiles on disk that maximize the efficiency of costly directory-access, file-open and block-read operations. By assigning diamonds a Sierpinski index, we can utilize this method to store tiles and retrieve tiles quickly.

Working with data sets too large to fit in-core requires efficient management of smaller subsets of the data to maintain application interactivity. An indexing scheme is needed such that each subset index is unique. Mapping the diamonds to Sierpinksi triangles guarantees this property for each diamond and also guaranteed spatial locality on disk. If the data is stored on disk in tiles according to their Sierpinksi index, most diamonds in a tile are spatially closer. A diamonds index is stored in 64-bits, where the upper bits represent the Sierpinksi index followed by a 1 and string of zeros to the end. See Appendix I for a full description of the mapping of Sierpinski indices to file names.

When a tile is requested, it is returned immediately if it is in cache. 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 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 height 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 replacement 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 so 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.

7 Rendering

The mesh triangulation is rendered after processing elements from the geometry and texture queues, respectively, taking into account approximation errors for both geometry and texture. For every frame, a fixed number of splits and merges are accomplished and a small number of textures are changed. Most of the geometry and textures remain fixed and are rendered from their cached versions.

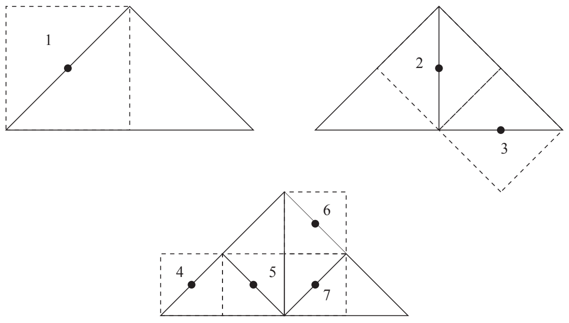

Figure 9: Sierpinski nodes associated with diamonds. The diagonal of a diamond can be associated with one edge of each isosceles right triangle in the bintree. With these associations, we can store diamond hierarchies using Sierpinski indices.

8 Results

Our performance results were measured from a 3Ghz Xeon processor with 1GB of RAM and a GeForce FX 5900 Ultra. We ran the demo at a resolution of 640 x 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 10-meter DEM data of Washington state with geometry and texture dimensions of approximately 111K x 137K. Textures were procedurally generated and colored from the original geometry and stored in RGB-565 format. To avoid the overhead of texture object memory allocation, we initialize a pool of texture memory at start-up and use TexSubImage calls to swap in new textures.

The out-of-core processing 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 rendering the high detail mesh. During this time, triangle chunks that need to be updated either due to geometry

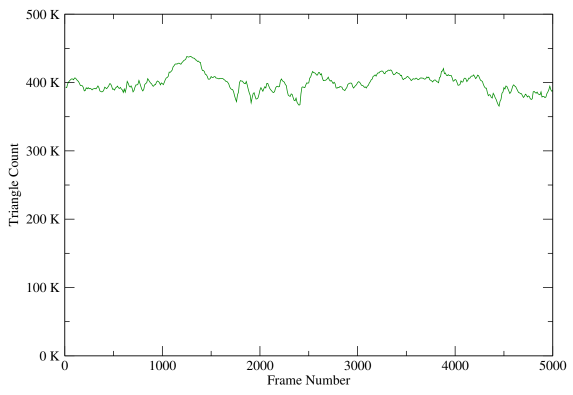

Figure 10: A plot of triangles-per-second (in millions) measured for each frame in the Washington state flyover.

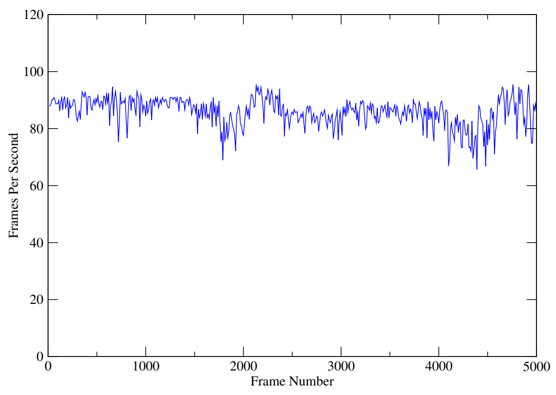

Figure 11: A plot of the frames-per-second during a portion of the Washington state flyover. Note the consistent frame rate even though the terrain changes dramatically.

updates or texture coordinate updates are transferred to AGP memory to be pulled by the GPU. Around 36% is spent traversing the hierarchy to evaluate when geometry chunks or uv texture coordinate updates are necessary. The time taken by the split/merge loop is a user defined parameter, but in this benchmark less than two percent time was spent. Less than one percent each was actually spent on fetching geometry and texture from disk, priority updates, UV calculations, triangle chunk 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. Our results show that the main bottleneck lies in the graphics card bandwith and the loop for determining appropriate triangle chunk updates to geometry and texture. Although our algorithm performance is not comparatively best in terms of speed, it offers superior image quality over quad-tree based systems due to our 4-8 scheme while still maintaining interactivity at high frame rates.

Video Description

The best way to evaluate this algorithm is to view the video. This video shows a short flight path from the northwest of Washington towards Mount Rainier. Benchmark statistics are overlayed at the top of the screen. We fade into three other modes to illustrate the work being done per frame. The wireframe mode depicts the triangle chunks being drawn and the granularity of the triangles in

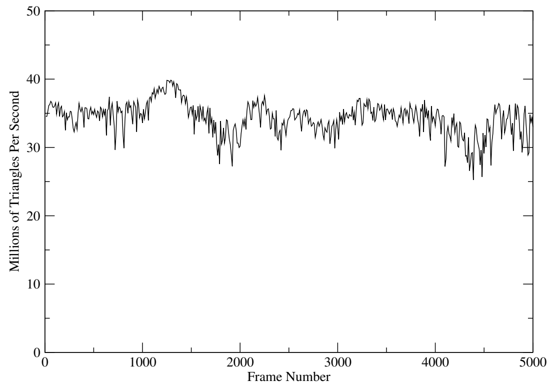

Figure 12: A plot of the number of textured triangles per second displayed for each frame in the Washington state flyover.

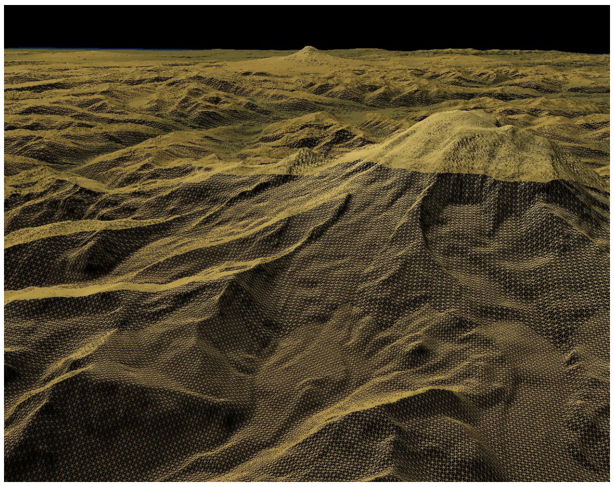

Figure 13: Wireframe mesh detail

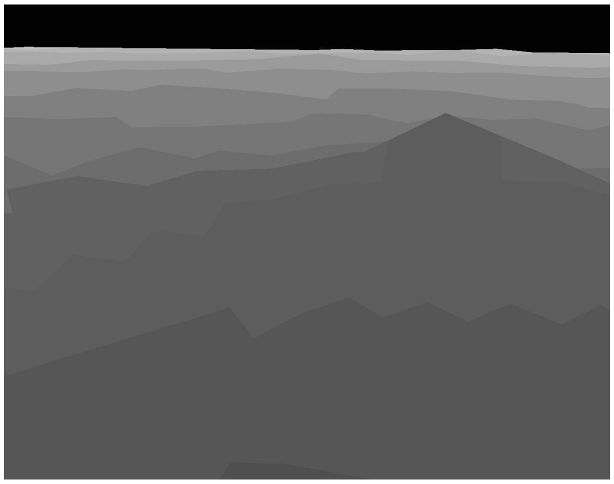

Figure 14: This is the same view as Figure 13 but shaded to show the texture level of detail independent of the geometry triangulation.

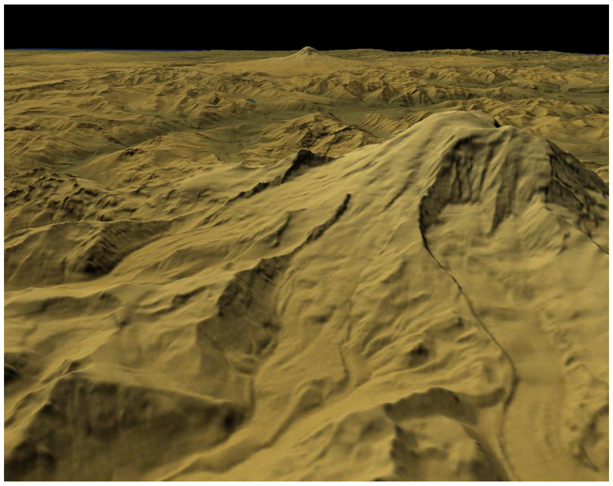

Figure 15: The Washington state flyover. Mount Rainier in the foreground and Mount Adams behind.

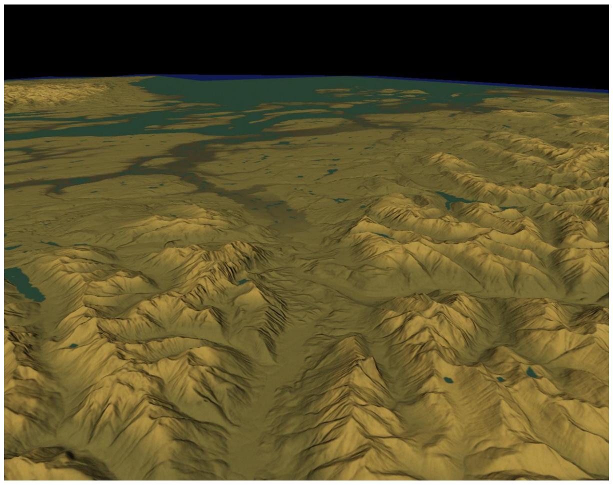

Figure 16: A view facing Victoria, Washington.

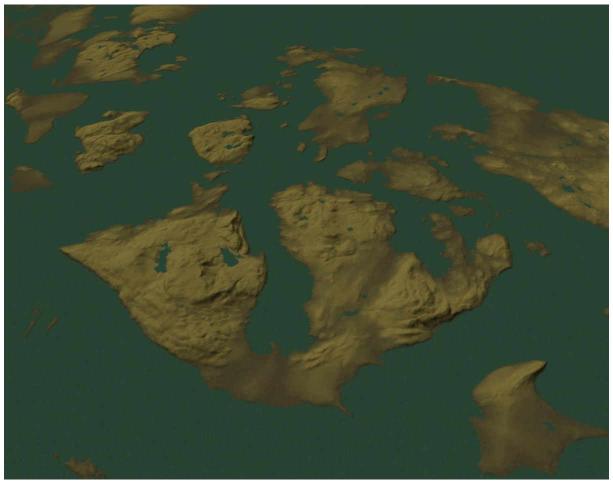

Figure 17: The San Juan Islands.

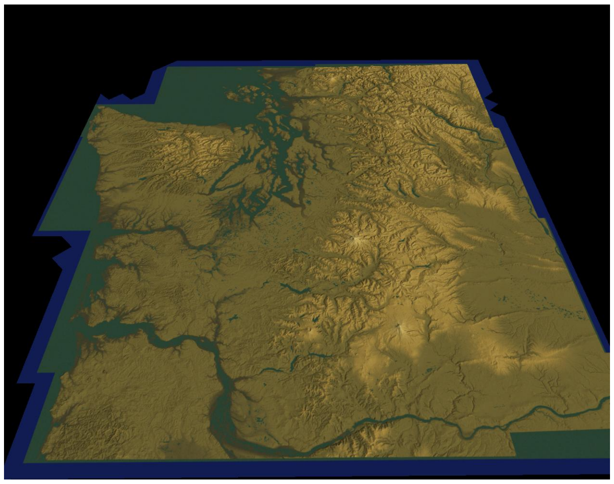

Figure 18: USGS Washington state terrain data set rendered using 4-8 texture hierarchies. Various texture levels of resolution are displayed on the mesh in a seamless tiling.

the the scene. A quick view of the underlying triangulation in wireframe is shown which fades into a shaded version of the mesh based on level of detail in the hierarchy, where the lighter shades have a lower resolution. No per-pixel blending of texture level-of-detail seems to be visible; we believe this is due to the gradual factor-of-two changes in information content between levels.

9 Conclusions and Future Work

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.

10 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), the Lawrence Livermore National Laboratory under contract B523818, and by Lawrence Berkeley National Laboratory.

References

[1] National Aeronautical and Space Administration. Mola data set, http://pds-geosciences.wustl.edu/missions/mgs/megdr.html, 2004.

[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 P. Brunet and D. Fellner, editors, *Proceedings of the 24th Annual Conference of the European Association for Computer Graphics (EG-03)*, volume 22, 3 of *Computer Graphics forum*, pages 505–514, Oxford, UK, September 1–6 2003. IEEE Computer Society, Blackwell Publishing Ltd.

[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*, Lecture Notes in Computer Science, pages 21–29. Springer-Verlag Inc., New York, NY, USA, 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 *Proceedings IEEE Visualization*, pages 147–155, Conference held in Seattle, WA, USA, October 2003. IEEE Computer Society, IEEE Computer Society Press.

[5] Jürgen Döllner, Konstantin Baumann, and Klaus Hinrichs. Texturing techniques for terrain visualization. In T. Ertl, B. Hamann, and A. Varshney, editors, *Proceedings Visualization 2000*, pages 227–234. IEEE Computer Society Technical Committee on Computer Graphics, 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 Roni Yagel and Hans Hagen, editors, *Proceedings of the 8th Annual IEEE Conference on Visualization (VISU-97)*, pages 81–88, Los Alamitos, October 19–24 1997. IEEE Computer Society, IEEE Computer Society Press.

[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 30 1997. Wed, 08 Jan 97 00:00:00 GMT.

[8] R. J. Fowler and J. J. Little. Automatic extraction of irregular network digital terrain models. *Computer Graphics (SIGGRAPH '79 Proceedings)*, 13(3):199–207, August 1979.

[9] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings of IEEE Visualization '98*. Institute of Electrical and Electronics Engineers, Inc., January 1998. Hoppe, H., "Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering," *Proceedings of Visualization '98* IEEE, Piscataway, NJ, 1998, pp. 35-42.

[10] D. E. Knuth. *The Art of Computer Programming, Sorting and Searching*. Addison-Wesley, Reading, MA, USA, 2 edition, 1975.

[11] Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In Robert Moorhead, Markus Gross, and Kenneth I. Joy, editors, *Proceedings of the 13th IEEE Visualization 2002 Conference (VIS-02)*, pages 259–266, Piscataway, NJ, October 27–November 1 2002. IEEE Computer Society.

[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 Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 109–118. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[13] Peter Lindstrom and Valerio Pascucci. Visualization of large terrains made easy. In Thomas Ertl, Ken Joy, and Amitabh Varshney, editors, *Proceedings Visualization 2001*, pages 363–370. IEEE Computer Society Technical Committee on Visualization and Graphics Executive Committee, 2001.

[14] 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.

[15] Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings Visualization 98*, pages 19–26,515, Los Alamitos, California, 1998. IEEE, Computer Society Press. extended version available as technical report ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/2xx/292.ps.

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

[17] United States Geological Service. State of washington data set, http://rocky.ess.washington.edu/data/raster/tenmeter/onebytwo10/index.html, 2004.

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

[19] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: A virtual mipmap. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 151–158. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.

[20] Thatcher Ulrich. Rendering massive terrains using chunked level of detail control, siggraph course notes, 2002.

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

[22] L. Williams. Pyramidal parametrics. *Computer Graphics (SIGGRAPH '83 Proceedings)*, 17(3):1–11, July 1983.

[23] Denis Zorin and Luiz Velho. 4–8 subdivision. *Computer Aided Geometric Design*, 18(5):397–427, 2001.

APPENDIX I

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:

- $b_{63}b_{62}b_{61}...b_N100...0$

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

This bit string can now be treating 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.