A Heightfield on an Isometric Grid

Morgan McGuire and Peter G. Sibley

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-05-14
October 2005

A Heightfield on an Isometric Grid*

Morgan McGuire† Brown University

Peter G. Sibley‡ Brown University

Abstract

The iso-heightfield is built on a grid with 60°-axes that produce an equilateral tessellation, which is the closest packed *regular* tessellation. We design a grid- to world-space mapping that is convenient for triangle strips and provides square tiles for level of detail and texture management. This makes the iso-heightfield an efficient drop-in replacement for the orthogonal heightfields commonly used today. We introduce several new support algorithms including fast vertex normals and self-shadowing. The iso-heightfield requires resampling when data was previously captured from an orthogonal grid or irregular samples. This is a lossless process if the original sampling filter is known and the Nyquist rate was observed.

Keywords: heightfield, terrain, hexagonal sampling, hardware rendering

1 Introduction

We describe a data structure for improved quality real-time rendering of large and dynamic heightfields on programmable graphics hardware. Our experiments show that the new *isometric heightfield* (iso-heightfield) gives a 10–25% more accurate representation of elevation, smoother vertex normals and shading, and smoother curvature than a traditional orthographic heightfield (ortho-heightfield). The rationale behind our use of an isometric grid is that isometric (or hexagonal) sampling is the optimal regular sampling pattern of scalar 2D circularly band-limited signals [Dudgeon and Mersereau 1990]. Our contributions, described below, are making isometric sampling practical for use in a modern rendering system; demonstrating that the optimal sampling pattern leads to improved curvature and shading in addition to elevation representation; and quantifying the advantages of an isometric heightfield at the relatively low polygon counts (sampling rates) used for practical applications in computer graphics. Our experiments include both objective and exhaustive tests on theoretically interesting meshes and subjective tests on specific meshes used for real-world applications like games and geology research in consultation with domain experts.

Although very good *irregular* and *semi-regular* tessellation methods like ROAM [Duchaineau et al. 1997] exist, heightfields remain popular for games and scientific visualizations. This is because the regular grid allows efficient on-disk storage and matches the sampling grid for dynamic data like simulated ocean waves, deformable terrain, and animated 2D functions. Regular tessellation is also well suited to hardware rendering because it is easy to triangle strip, the memory footprint is fixed, and it is easy to change the level-of-detail (LOD) via Geometrical MipMapping [de Boer 2000] (geomipmapping).

An ortho-heightfield tessellates a surface by splitting Cartesian grid squares along a consistent diagonal. This forms right triangles in the xz-plane that are distorted by their y-elevations. The perpendicular to the diagonal axis becomes a preferred orientation for features. Where the surface curves parallel to the diagonal the heightfield is a sub-optimal approximation of the true surface. As shown by the shaded mesh and Gaussian curvature in Fig. 1(left), this gives a misleading impression of roughness. The sine wave in the figure is relatively low-frequency compared to the tessellation; high-frequency features like ocean waves in a game and glacial ripples in a geology data set can degenerate so far as to appear to be individual spikes instead of waves.

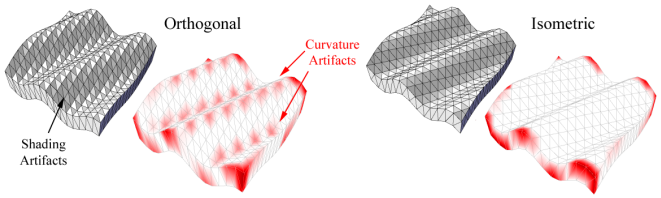


Figure 1: The iso-heightfield minimizes the curvature and shading artifacts that make an orth-heightfield appear rough even at low frequencies.

The ortho-heightfield can be improved by choosing the splitting diagonal independently for each square [Akenine-Moller and Haines 2002]. Unfortunately, a consistent diagonal is needed for triangle strips and efficient elevation interpolation. For dynamic data, recomputing the ideal diagonal every frame can grow expensive, and switching diagonals between frames creates jumping artifacts.

An overriding concern for real-time rendering of any 3D surface is the interaction between CPU and graphics processing unit (GPU). Today's GPUs are substantially more powerful than CPUs and are isolated behind a PCI express or AGP bus that is slow compared to the processor throughput. To achieve high rendering performance, we store data in graphics memory and place most rendering computation to the GPU. The CPU needs to access only a small amount of data per frame and we avoid the CPU/bus bottleneck. So-called SM3.0 graphics hardware is just emerging that can read texture memory from the vertex stage. On this hardware it is convenient to store terrain elevations in a texture, instead of geometry as we have done. Unfortunately few texture reads are allowed per vertex for the cards commercially available today (e.g., NVIDIA GeForce6800) and those accesses are both slow and unfiltered. Therefore, we give preferred implementations that efficiently process elevations packed into the texture attribute of a vertex and note how the preferred implementations of our algorithms differ for future hardware where it is non-obvious (e.g., vertex normal computation).

Conceptually, a heightfield is a trivial data structure. Yet the support algorithms— the mapping between coordinate frames, triangle strip ordering for multiple levels of detail,

*An earlier version of this work was presented with the same title in the SIGGRAPH 2004 Sketches program.

†morgan@cs.brown.edu

‡pgs@cs.brown.edu

sampling, and vertex normal computation — reveal interesting geometric relationships and create competing design constraints. It is important to note that although the individual algorithms are *derived*, the data representation and implementation presented in this paper are *designed*, as they are for any other data structure. We created the iso-heightfield from a desire to unify, and to optimize for both performance and fidelity, the heightfields we had implemented in other visualization and real-time graphics research projects. It is a compromise that we believe is a local maximum in the design space and is intended as a good all-around structure for visualizing 2D functions.

Our contributions are a new coordinate mapping, the derivation of a set of consistent support algorithms, a data representation for modern programmable hardware, and experimental results that show the iso-heightfield more accurate than an ortho-heightfield on average. We complement the mathematics with pseudo-code and diagrams to aid implementors, particularly since it is easy to make "off-by-one" errors when computing the indices that figure prominently in our algorithms.

All indices are zero based. Throughout we use $(column, row)$ indexing to match the (x, z) diagrams. Although tiles can have any orientation in world space and need not represent terrain, for convenience we call the $z = 0$ edge of a tile the "North" and orient diagrams so it is at the top. The $x = 0$ edge is drawn on the left and called the "West", and so on.

1.1 Related Work

Musgrave [1998] originally noted the smoothness of equilateral tessellation and proposed creating one for ray-tracing by shearing a tile of an orthogonal heightfield to form a parallelogram. Because of their symmetry, hexagonal grids have also been proposed for both texture synthesis [Middleton 2002] and image processing [Middleton and Sivaswamy 2002]. Hexagonal sampling has also been explored in the context of rendering and processing volumetric data [Neophytou and Mueller 2002; Theussl et al. 2001]. We adapt Musgrave's idea for real-time use by creating an indexing scheme that allows square tiles, computing per vertex normals, and reducing the memory requirement. Square tiles are necessary for mapping square textures over tiles without distortion, and are more practical for paging in parts of a large world, working with LOD, and interacting with existing square-grid infrastructure for object and zone management.

Quite a few large scale terrain visualization methods have been proposed. ROAM [Duchaineau et al. 1997] uses triangulated irregular networks(TIN) and adaptive view dependent simplification. Hoppe [1998] developed similar methods in the context of view dependent progressive meshes. Pajarola [1998] uses TINs derived from restricted quadtree-based triangulations. Cignoni et al. [2003] have exploited programmable graphics hardware for TINs. Losasso and Hoppe [2004] use view dependent down sampled versions of an ortho-heightfield combined with a real-time GPU decompression scheme. Our iso-heightfield is complementary to Lossasso and Hoppe's work and both leverage regularity of the heightfield for simplification and efficient hardware accelerated algorithms. Several excellent schemes exist for level-of-detail simulation on regular grids [Lindstrom et al. 1996; Levenberg 2002; de Boer 2000]. We show how to use a hardware friendly variant of geomipmapping [de Boer 2000].

Shankel [2002] presents an algorithm for per-vertex normals on an ortho-heightfield. We extend this to consider the six equal faces that meet at an iso-vertex.

Shadow casting for terrain is a challenging problem both because it is huge in extent and shadowing is a non-local effect, and because the meshes are so highly tessellated. The first property defeats LOD and culling schemes since a far-off peak may cast a shadow on nearby objects. The second defeats brute force approaches and demands a sub-linear time solution to an inherently quadratic (i.e., all pairs) occlusion problem. General purpose perspective shadow maps and stenciled shadow volumes offer approximately linear time shadowing, but have tremendous fill rate requirements (even with optimizations for modern graphics cards [McGuire 2004b; Kozlov 2004]) that make them impractical for the meshes with hundreds of millions of polygons that we consider.

We propose a new method inspired by the geometry of shadow volumes that provides real-time shadow casting for *dynamic* terrain and movable light, when the light is at infinity and moves strictly along the x-axis. Like light maps, we compute shadowing at a low frequency and filter to create fake penumbrae, and reuse shadow results for several frames. Our method is linear in the number of *viewer-visible* polygons and the time cost is amortized over the frames for which results can be cached. Although our method is neither general nor physically correct[1], it is extremely practical and is well-suited to real-time visualization and games.

2 Experiments

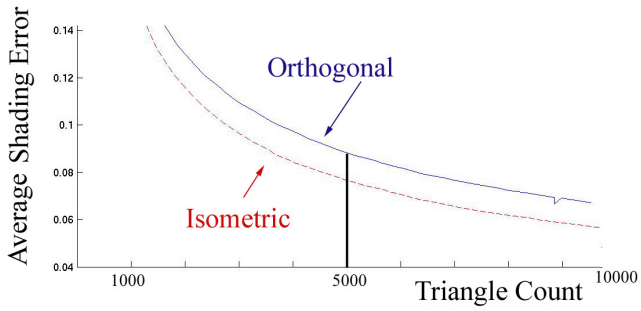


Figure 2: The Isometric- and Ortho-heightfield mean shading error. We averaged over orientation of the sampling grid and over the entire range of measured frequencies.

We performed several experiments to determine the reconstruction error from samples over an isometric and orthogonal grid. Choosing the test heightfield for measuring reconstruction error poses an interesting problem. One can look at meshes actually used in real-world applications (as we do later in Section 4). However, working with specific meshes biases the result towards those meshes and makes no statement about the generality of the data structure. We view terrain or any other function to be represented by a heightfield as a continuous, scalar-valued signal on the plane. The heightfield data structure stores a filtered and sampled version of that signal from which it attempts to reproduce the original. From this viewpoint it is natural to borrow a technique from signal processing to analyze the fidelity of reconstruction.

[1] Even the Sun does not move in a perfect East-West arc except at Earth's equator on the solstices.

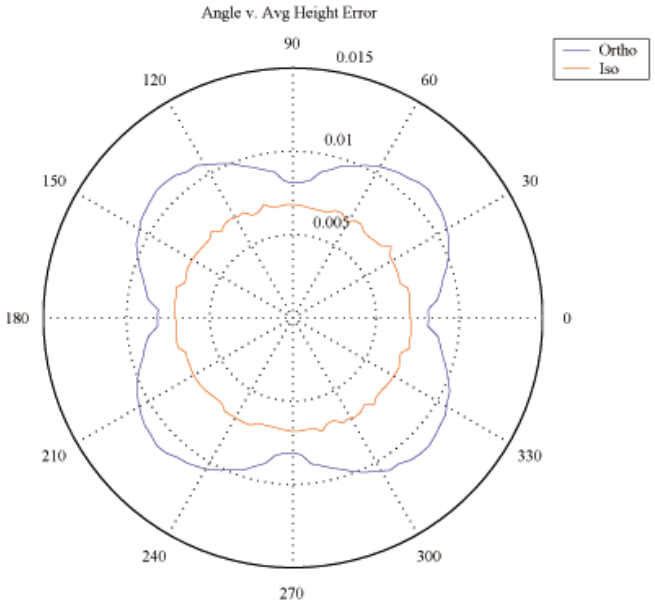


Figure 3: Mean elevation error versus the angle of the sampling grid for a sine function with frequency 0.9.

Using Fourier analysis, one can decompose any continuous function into a weighted sum of sinusoidal functions of differing frequencies. A heightfield capable of representing 2D sine waves of *any* frequency and orientation with high fidelity is likely to also represent sums of those waves with high fidelity. For our analytic experiments we created a series of 22,800 different sine waves. These waves varied all combinations of: phases from 0 to *pi* in increments of $\pi/12$, orientations from 0 to 2π in increments of $\pi/50$, and frequencies from 0.01 to 1.00 in increments of 0.05.

We fixed the regular grid of (xz) points to sample, then used conjugate gradient descent to optimize y values to minimize the error between the regularly sampled heightfield and the source sine function. This was done for both orthogonal and isometric grid patterns. Once the two heightfields were created, we measured the mean elevation error, the root mean square elevation error, the average error of normals, and the average shading error. Shading error is what viewers primarily perceive; the elevation error is more significant for physical simulation.

In each case, error is measured between the reconstructed function and the original analytic function that served as ground truth. We exhaustively measured the error for each sine wave in the series for meshes with polygon counts between 1,000 and 10,000, neglecting only experiments where the polygon count was below the Nyquist sampling rate for the given sine wave frequency. We believe that the hundreds of thousands of trials performed adequately measure the reconstruction error behavior of the heightfields.

We observed that reconstruction error was insensitive to phase, so all reported errors are averaged over all measured phases. Reconstruction error varies with respect to orientation and polygon count.

Fig. 2 plots mean shading error versus polygon count. The iso-heightfield has 25% less error than the orthogonal heightfields at the common 5000 polygon case.

Fig. 3 plots mean elevation error versus sampling grid angle on polar axes. The iso-heightfield produces less error and is less dependent on the orientation of the sampling grid. Notice the error of the orthogonal grid (blue) has four-fold symmetry, while the isometric grid has strictly lower error at all orientations and is resistant to orientation changes.

The graphs for the other metrics applied are nearly identical so we do not reproduce them here.

Table 1: Sample tile dimensions for constrained number of levels of detail Λ and visible triangle count $|T_\lambda = 0|$ that give nearly equilateral triangles in the xz plane.

Λ	$\lvert T_{\lambda=0}\rvert$	C	R	Aspect
1	6	5	5	0.866 (Fig. 4)
	1k	24	29	1.046
	5k	49	57	1.007
3	136	12	11	0.794 (Fig. 6)
	1k	24	27	0.974
	5k	48	55	0.992
4	6k	52	67	1.116
	10k	68	83	1.057
	50k	156	179	0.994
	100k	212	243	0.993
10	15M	2564	3075	1.039 (Fig. 8)

3 Data Structure

We subdivide the entire heightfield into *tiles*, each of which can be rendered with a separate LOD and texture. At the highest detail level ($\lambda = 0$), a tile contains $C \times R$ elevation values and $|T_{\lambda=0}| = (2C-7)(R-3)$ visible triangles. The odd constants appear because each tile is padded with a one-vertex border for computing vertex normals at the edge.

The choice of R and C is highly constrained. So that tiles meet their North and South neighbors at vertices, R must be odd. For the triangles to be nearly equilateral $R \approx C/\sin\frac{\pi}{3}$, where $C < 2^{14}\sqrt{3}$ so that tile grid coordinates fit the range of GL_SHORT vectors. For Λ detail levels (including full detail), both $(C-4)$ and $\frac{1}{2}(R-3)$ must be integer multiples of $2^{\Lambda-1}$.

Table 1 lists values of C and R that satisfy these constraints. We generated the values for each row by choosing Λ and N_v and then searching for the integers i and k that minimized aspect ratio distortion, $|R\sin\frac{\pi}{3}/C - 1|$, where $R = i2^{\Lambda-1}+3, C = j2^{\Lambda-1}+4$. The $|T_{\lambda=0}| = 5000$ rows of the table are interesting because that is the efficient triangle strip length for today's graphics cards. The *aspect* column gives the ratio of $R\sin\frac{\pi}{3}/C$, which is unity for the ideal tile.

3.1 Coordinate Mapping

Unlike Musgrave's uniform shear, our mapping slides odd rows horizontally by 1 unit in object-space. It also slides border vertices by 1 unit on even rows (black circles in Fig. 4) to form a few isolateral triangles that keep the East and West tile edges straight.

As with an ortho-heightfield, elevations are stored in an array where $e[c + rC]$ is the height at integer location (c, r) in a conceptual *grid space* (Fig. 4a). The corresponding *object-space*(Fig. 4b) vertex is:

$$(x, y, z) = (2c + (r \text{ BITAND } 1) + b - 3, e[c + rC], r - 1) \quad , \quad (1)$$

where $b = 1$ when $c = 1$ and r is even, $b = -1$ when $c = C-2$ and r is even, and $b = 0$ otherwise. Multiplying object-space

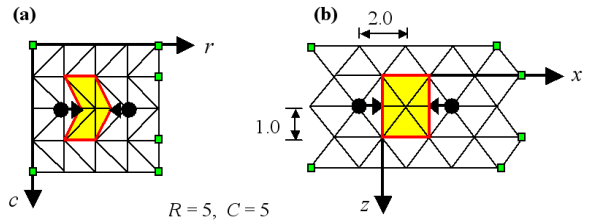


Figure 4: (a) Grid Space (b) Object Space

vertices by the scaling matrix σ transforms them to *world-space*.

$$\sigma = \begin{pmatrix} \frac{extent}{2(C-4)} & 0 & 0 \\ 0 & y_{max} & 0 \\ 0 & 0 & \frac{extent}{R-3} \end{pmatrix} = \begin{pmatrix} \sigma_x & 0 & 0 \\ 0 & \sigma_y & 0 \\ 0 & 0 & \sigma_z \end{pmatrix} \quad (2)$$

In world-space, the triangles are equilateral, the tile is a square with sides of length *extent*, and elevations are in the range $[0, y_{max}]$. A translation by integer multiples of *extent* positions tiles within the full heightfield.

Recall that each tile has a one-vertex border for computing normals. In the figure, the red outline shows the actual edges of the tile when rendered. Six vertices (shown as green squares) are unused for either rendering or normal computation But they remain in the elevation array to preserve indexing and data alignment. Note that the smallest possible tile is shown for demonstration purposes; for reasonably large tile sizes most vertices are on visible triangles.

3.2 Vertex Compression

Each vertex in video memory is compressed to two bytes and reconstructed on programmable hardware using the vertex processor. The (x, z)-values are the same for each tile and are stored in a single vertex buffer shared among all tiles. Each tile's y-values are in their own buffer. The coordinate mapping is designed so that all components are encoded as 16-bit integers, although the y-values can be at different bit-resolution because they are in a separate buffer. For procedural terrain the y-values of the heightfield are not stored.

3.3 Rendering

Careful indexing allows us to render each tile with a single triangle strip. This reduces the size of the index array (which is also shared among all tiles) and optimizes the vertex cache performance on graphics hardware.

The appendix gives the details of the algorithm. Lines of code are colored to match the triangles depicted in Fig. 6 that they produce. The triangle strip begins at the most North-East visible vertex in grid-space, $(C-2, 1)$ for zero-based indexing. It alternates winding North-West and South-West (red row, following the arrows) until the West edge is reached. That vertex is repeated twice (green dots) to create a degenerate triangle and set up for the next row. The degenerate triangle is needed to keep the face winding direction consistent between rows. Next, the strip winds South-East and North-East (blue row) to the East edge. That vertex is repeated, and then another two vertices *two* rows down are added to form another degenerate triangle. The entire process then repeats. In total there are $(R-3)(2C-3)-3$ indices forming what we call the *full detail* mesh. It is rendered by setting the hardware vertex pointer to the vertex buffer and sending the index array with OpenGL `gl_DrawArrays` or DirectX `DrawPrimitive`.

When the projected dimensions of its triangles fall below the size of a pixel because a tile is far from the viewer, it can be rendered from a lower detail mesh. We build discrete levels of detail numbered $\lambda = 0 \ldots (\Lambda-1)$ using the isometric equivalent of geomipmapping [de Boer 2000] and propose a method for disguising visible artifacts that would otherwise occur when the detail level changes.

The vertex buffers containing the grid and elevations are identical for each level of detail. Only the index arrays specifying the traversal order of those vertices change. Each detail level is described by a *strip array* of indices that creates a triangle strip from the elements of the vertex buffer and a *stitch array* that creates a list of triangles from the same buffer. The strip covers the main body of the tile and the stitch prevents T-junctions at tile borders between differing detail levels.

Because the index arrays are precomputed, the LOD can change without uploading any new data to the graphics card. There is also no additional space cost per-tile for the additional detail levels because they work from a single elevation array.

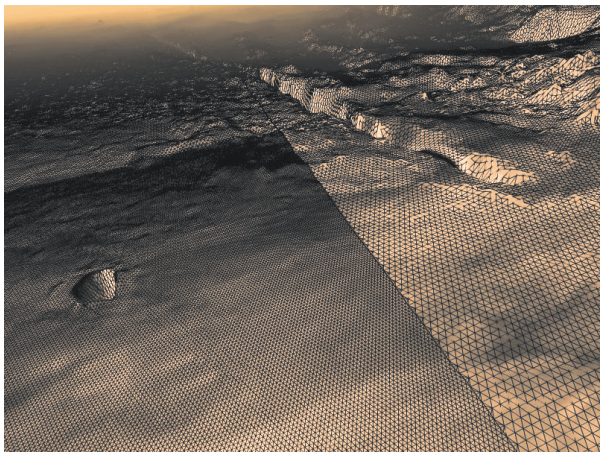


Figure 5: The interface between two detail levels.

3.4 Computing the Strip Array

The full detail, $\lambda = 0$, mesh winds through every visible vertex in the switchback pattern described previously. The next lower detail mesh, $\lambda = 1$, contains approximately 1/4 as many triangles and is created by winding through vertices from alternating rows and columns. In general, we follow the level 0 pattern but step between grid points with stride of 2^λ rows and columns. The indexing is slightly complicated by boundary cases and by peculiarities of level 0. Recall that even and odd rows are shifted relative to one another. Only level 0 touches vertices on even rows, so the algorithm for level 0 even row indices contains an additional "+1" that does not appear for other detail levels.

The row loop in the code generates triangles for two rows and there are $\frac{R-3}{2^\lambda}/2$ *pairs* of rows of triangles. Each row contains two triangles per column (the "half triangles" each count as one), with the first one and last two columns omitted as padding. A triangle strip contains two vertices plus one per triangle, so each row contains $2\frac{C-4}{2^\lambda}+2$ vertices and each *pair* of rows contains twice that amount. There are

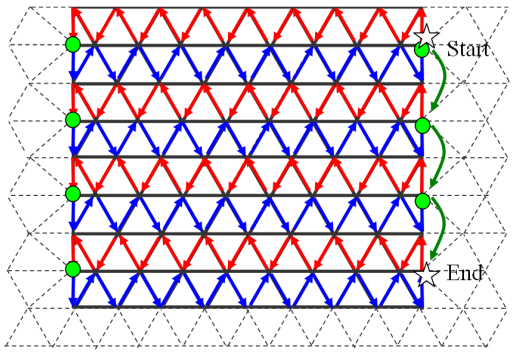


Figure 6: Triangle strip vertex ordering ($C = 12, R = 11, \lambda = 0, \Lambda = 3, |T_{0,strip}| = 136, |I_{0,strip}| = 165$)

six additional vertices per *pair* of rows: two per red row and three per blue row that are degenerate (green), and one setup vertex for each red row. In total there are (pairs of rows) × [(vertices per row pair) + (additional vertices per row pair)] minus three degenerate vertices that are not needed for the final row. The size of the index array for the triangle strip at level λ is:

$$|I_{\lambda,strip}| = \frac{R-3}{2^{\lambda+1}}\left(\frac{C-4}{2^{\lambda-2}} + 10\right) - 3\,. \tag{3}$$

Since the loops must execute an integer number of times, $\frac{1}{2}(R-3)$ and $C-4$ must be integer multiples of $2^{\Lambda-1}$.

The number of visible (non-degenerate) triangles at detail level λ is:

$$|T_{\lambda,strip}| = \left(\frac{C-4}{2^{\lambda-1}} + 1\right)\frac{R-3}{2^{\lambda}} \tag{4}$$

3.5 Computing the Stitch Array

There are two ways stitch adjacent tiles with different levels of detail: each of the four tile edges (North, South, East, West) can precisely stitch the current tile to its counterpart, or each tile's edges can be stitched to a common detail level so that all neighbors are automatically compatible. We choose the second method, stitching each level to the borders of the full-detail mesh. This allows tiles to be rendered independent of their neighbors and reduces the number of rendering calls from five to two per-tile. The latter is important because only a few hundred draw calls can be made per frame (regardless of the number of vertices) in the DirectX API before a program becomes CPU bound [Harris 2004], a limitation that game developers particularly encounter with terrain rendering [McGuire 2004a]. This problem does not occur in the OpenGL API, which trades off inexpensive draw calls against comparatively expensive state changes.

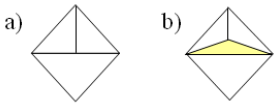


Figure 7: a) T-junction b) Resolved by adding one triangle (yellow), shown with an exaggerated crack.

Where detail level k meets level $k+1$ there is one T-junction per border triangle. Combined, the North and South borders contain $2\frac{C-4}{2^{\lambda}}$ border triangles and the East and West borders contain $2\frac{R-3}{2^{k}}$ together. As shown in Fig. 7, each T-junction can be resolved by adding one triangle (yellow) that joins the corners to the center. Therefore, to resolve all T-junctions between these sequential detail levels we must add $\frac{R+C-7}{2^{k-1}}$ triangles to the list. To resolve all T-junctions between level 0 and level λ we repeat the process λ times. The number of visible triangles and size of the index array for the stitch list at level λ are thus:

$$|T_{\lambda,list}| = \begin{cases} 0 & \lambda = 0 \\ (R+C-7)\left(2-2^{1-\lambda}\right) & \lambda > 0 \end{cases} \tag{5}$$

$$|I_{\lambda,list}| = 3|T_{\lambda,list}| \tag{6}$$

The appendix gives the precise indices. Because the stitch list for a level is a super-set of the stitch list for the previous level, we store a single list for all detail levels and pass a pointer to the beginning and the appropriate length to rendering calls.

Because we chose to always stitch the boundary to the full detail mesh, after several iterations the total triangle count $|T_\lambda| = |T_{\lambda,strip}| + |T_{\lambda,list}|$ is dominated by the stitch triangles. Fig. 8 shows this point of diminishing returns for a huge (by today's standards) tile. For this heightfield, further simplification yields small savings after five detail levels.

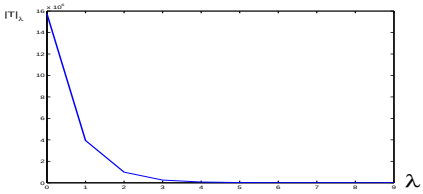


Figure 8: Total triangle count versus level of detail. $C = 2564, R = 3075$

3.6 LOD Transitions

We can not abruptly change the level of detail for a single tile between sequential frames or several well-known LOD artifacts collectively known as "popping" would be visible. These artifacts are: the immediate change in elevation of vertices, the change in shading, and the change in texture Mip-map level from the triangle angle changing. De Boer [de Boer 2000] likens LOD popping to texture popping when using discrete Mip-map levels and proposes morphing vertices between levels. Morphing requires either changing the vertices per-frame or storing redundant data per-vertex.

To reduce popping without modifying the vertex data during transition, we alpha blend in detail levels over multiple frames. The challenge is that we want to blend the *images* of the old and new detail levels, but cannot do so without render-to-texture (which introduces latency) and careful sorting at boundaries where adjacent tiles overlap in screen space. We propose a rendering scheme that achieves a similar effect with greater efficiency.

3.7 Shrinking Filter Systems

In the most general case, a *coarse* (low-detail) version $C(x)$ of a real-valued function is obtained from a *fine* (high-detail)

version $F(x)$ by low-pass filtering and decimating. The surface between samples of the coarse version is then interpolated using a reconstruction filter. The collective processes of low-pass filtering, decimating, and reconstructing are a filter system. We classify a system as *shrinking* if, for every location between two sample locations, $x_1 < x < x_2$:

$$C(x) \geq F(x) \text{ where } \frac{d^2F(x)}{dx^2} < 0 \text{ and}$$
$$C(x) \leq F(x) \text{ where } \frac{d^2F(x)}{dx^2} > 0 \text{ and}$$
$$\frac{d^2F(x)}{dx^2} \neq 0.$$

In general, the shrinking property means that the fine level has strictly higher elevations than the coarse level in convex areas and strictly lower elevations in concave areas. This is true for many, but not all filter systems[2]. For our LOD scheme, the low-pass filter is an impulse (i.e. we do not filter), decimation is by a factor of four, and the reconstruction filter is Barycentric. We use the shrinking property of this system to our advantage by smoothing LOD transitions using alpha blending.

3.8 Two-Stage Transition

Let λ be a real number where $F = \lfloor \lambda \rfloor$ is the next-finer detail level, $C = F+1$ is the next-coarser detail level, and $\beta = \lambda - F$ is the fractional detail. We classify a fractional detail level as belonging to one of two transition stages, depending on whether it is closer to the fine or coarse mesh. For a visually smooth transition, we alpha blend between the two detail levels. The stage determines the order of rendering and the α value:

$$\text{Near-Fine}(\beta < \frac{1}{2}) : \text{Let}\alpha = 2\beta, L_1 = F, L_2 = C \quad (7)$$

$$\text{Near-Coarse}(\beta \geq \frac{1}{2}) : \text{Let}\alpha = 2 - 2\beta, L_1 = C, L_2 = F \quad (8)$$

The rendering algorithm is then:

1. Disable Alpha Blend; Enable Depth Write
2. Draw Detail Level L_1
3. Enable Alpha Blend; Disable Depth Write
4. Draw Detail Level L_2

For the near-fine stage, this blends fine to coarse in concave regions by drawing fine first, and then fading coarse in second. That leaves concave regions coarse and convex regions fine. For the near-coarse stage the algorithm blends fine to coarse in the convex regions by drawing coarse first, and then fading fine out second. In each case the depth buffer is set from the most visible stage. Note that we alpha blend from zero to one *twice* during the complete transition, hence the 2β terms.

[2]Systems in which the low-pass and reconstruction filters are normalized typically have the shrinking property. Zeroth order B-splines (nearest neighbor interpolation) do not have the shrinking property, but first (linear interpolation) and third order (cubic) B-splines do.

3.9 Selecting a Fractional Detail Level

Let A_w be the estimated world-space area of a typical $\lambda = 0$ triangle. This is bounded below by $\sigma_x\sqrt{3}/8$ (recall that σ_x is twice the world-space edge length) on a flat surface and bounded above only by the vertical resolution. The worst case occurs for a maximum curvature edge, like a cliff. For estimation purposes, we assume a surface at a 45-degree angle to the horizontal so the triangles have area given by $A_w = \sigma_y\sqrt{2}/4$. We convert this to a screen space value A_s by multiplying by the inverse of the camera-space z-ordinate of the center of the tile.

This process over-approximates the screen space area of a steep slope with the view vector opposite to the surface normal. The surface will typically be viewed from a glancing angle and is often shallow, so A_s is a fairly conservative estimator of screen space area.

Few detail levels are needed because level λ is used only when full-detail triangles have area $4^{1-\lambda}$ pixels– at level $\lambda = 5$, the triangle area is 1/256 of a pixel and the tile is likely to be well beyond the far clip plane.

3.10 Sampling Between Vertices

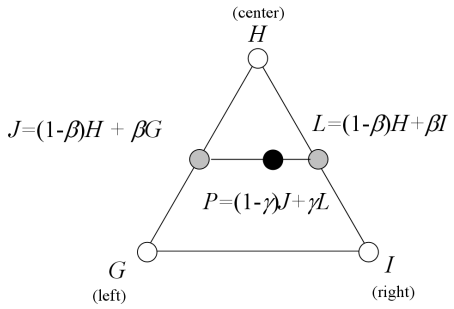


Figure 9: Geometric construct for interpolation

Elevation is a per-vertex property of a heightfield. One can imagine other per-vertex properties like color, material coefficients, and alpha. It is often useful to interpolate such properties between vertices in order to sample non-integer locations in grid-space. To satisfy this, we describe how to find indices g,h, and i of the surrounding triangle vertices G, H , and I for the point P at (non-integer) grid space location (c, r) and the Barycentric weights w_G,w_H, and w_I necessary for interpolating a property.

We proceed from the world-space vertex to three property indices and weights in three steps. First we find the tile containing the vertex. Second we find the vertex indices describing the triangle containing the vertex in the given tile. Third, we solve for the weights.

The first step is outlined in section 3.12. The second step has four cases to consider. The point (c, r) may be in either an *even* or an *odd* row of triangles, and in each row it may be in a *down* or *up* triangle. Fig. 10 shows the shape of these triangles. To determine which case we are in, we need to examine the row number and the fractional parts of c and r. Let u be the fractional part of c, v be the fractional part of r, and p be the vertex index corresponding to the integer parts of c and r: $u = c - \lfloor c \rfloor$, $v = r - \lfloor r \rfloor$,$p = \lfloor c \rfloor + \lfloor r \rfloor C$. Note that P does not correspond to a vertex, so p is not "the index of P" but instead the index of a nearby vertex.

Table 2: Cases for determining Barycentric weights.

Case 1:	Even Row, Down Triangle
$(\lfloor r \rfloor \mod 2 = 0)$ and $(u < 1 - v)$	
$g = p$	$h = p + cols$
$\beta = 1 - v$	$\alpha = u$
Case 2:	Even Row, Up Triangle
$(\lfloor r \rfloor \mod 2 = 0)$ and $(u \geq 1 - v)$	
$g = p + cols$	$h = p + 1$
$\beta = v$	$\alpha = u + v - 1$
Case 3:	Odd Row, Down Triangle
$(\lfloor r \rfloor \mod 2 = 1)$ and $(u > v)$	
$g = p$	$h = p + cols + 1$
$\beta = 1 - v$	$\alpha = u - v$
Case 4:	Odd Row, Up Triangle
$(\lfloor r \rfloor \mod 2 = 1)$ and $(u \leq v)$	
$g = p + cols$	$h = p$
$\beta = v$	$\alpha = u$

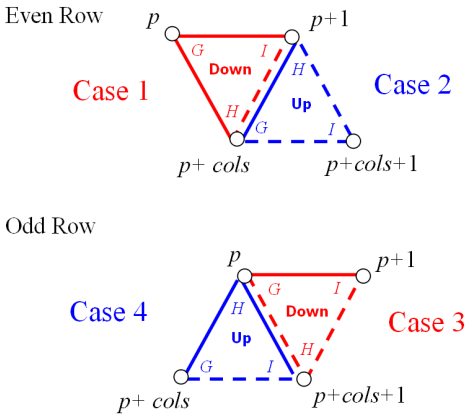


Figure 10: Depiction of cases from Table 2

We order the vertex indices of the triangle surrounding P so that G is the leftmost point, I is the rightmost point, and H is between them. The indices of g and h (i is always $g+1$) and parameters β and $\alpha = \gamma\beta$ for each case are given in the equations of Table 2. These follow from the geometry in Fig. 9. The third step computes weights from the parameters. It mimics Gouraud interpolation by finding endpoints of a horizontal line segment containing P and interpolating between them. The vertex weights, derived from the relation

$$P = w_G G + w_H H + w_I I = (1-\gamma)((1-\beta)H + \beta G) + \gamma((1-\beta)H + \beta I),$$

are given in Equation (9).

$$w_G = \beta(1-\gamma) = \beta - \alpha, \qquad w_H = 1 - \beta, \qquad w_I = \beta\gamma = \alpha \tag{9}$$

3.11 Per-vertex Normals

Per-vertex surface normals are needed for rendering smooth per-vertex lighting. First, this section derives a compact expression for the object-space surface normal at a vertex. Second, this section presents an efficient vertex program for computing world-space normals.

We derive an expression for object-space surface normals at point P. Let $P = (P_x, P_y, P_z)$ with neighbors $D \ldots I$ shown in Fig. 11. Let the indices of these vertices be p and $d \ldots i$. By the grid coordinate system, these have the relationship:

$$\begin{array}{lll} d = p - C - 1 + s & e = p - C + s & f = p + 1 \\ g = p + C + s & h = p + C - 1 + s & i = p - 1, \end{array} \tag{10}$$

where $s = \lfloor \frac{p}{C} \rfloor \mod 2$ accounts for the alternate-row shift. We define the normal at P to be the area-weighted average of the face normals from triangles PED, PFE, PGF, PHG, PIH, and PDI. That is, the vertex normal is the unit vector in the direction of:

$$N = PE \times PD + PF \times PE + PG \times PF + PH \times PG + PI \times PH + PD \times PI.$$

We choose the area-weighted normal because dividing out

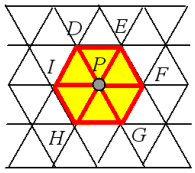


Figure 11: Clockwise neighbors of P: D, E, F, G, H, and I

face areas is computationally expensive. This is a reasonable choice; Max [1999] showed experimentally that area weighting is superior to both unweighted normals and the commonly used angle weighting proposed by Thurmer and Wuthrich [1998].

Substituting the individual edge vectors reduces the previous expression to:

$$PD = (-\frac{1}{2}, D_y - P_y, -\sin\frac{\pi}{3}) \qquad PE = (\frac{1}{2}, E_y - P_y, -\sin\frac{\pi}{3})$$
$$PF = (1, F_y - P_y, 0) \qquad PG = (\frac{1}{2}, G_y - P_y, \sin\frac{\pi}{3})$$
$$PH = (-\frac{1}{2}, H_y - P_y, \sin\frac{\pi}{3}) \qquad PI = (-1, I_y - P_y, 0)$$

$$N = (\frac{1}{6}\sin\frac{\pi}{3}[D_y - E_y + 2I_y - 2F_y + H_y - G_y], 1, \frac{1}{4}[D_y + E_y - H_y - G_y]).$$

Combining common subexpressions and scaling by 4 gives a compact expression for the (non-unit) normal. Its efficiency is important when working with dynamic data.

$$\text{Let } GD_y = D_y - G_y, EH_y = H_y - E_y, FI_y = I_y - F_y$$
$$N = (\frac{4}{6}\sin(\frac{\pi}{3})[GD_y + EH_y + 2FI_y], 4, GD_y - EH_y) \tag{11}$$

This is essentially a hexagonal version of the square-based fast normal method proposed by Shankel [Shankel 2002].

On a SM3.0 graphics card, normals can be computed in the vertex processor. We precompute the neighbor indices and store then in a texture, these are the same for every tile. We compute object-space normals with the following GLSL [Rost et al. 2004] vertex shader:

```
const vec3 k = (4/6)*sin(pi/3)*vec3(1, 1, 2);
vec3 A(tex2D(d), tex2D(h), tex2D(i));
vec3 B(tex2D(g), tex2D(e), tex2D(f));
vec3 C = A - B;
vec4 N1(dot(C, k), 4, C.x - C.y, 0);
```

As with any surface normal, `N1` can be transformed to world-space by the inverse transpose of the model-view matrix. In the shader, we compute `vec4 N = mul(MVinvT, N1)`, where `N` is the world-space normal. The above vertex program for computing world-space normals can be optimized as follows. Let M be the model view matrix. As a preprocess, we compute:

$$K = (M^{-1})^T \begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{2}{\sqrt{3}} & 0 \\ 0 & 0 & 0 & 4 \\ 1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} .$$

Using K as a `const uniform mat4` parameter, we use the following GLSL program:

```
vec3 A(tex2D(d), tex2D(h), tex2D(i));
vec3 B(tex2D(g), tex2D(e), tex2D(f));
vec4 C(A - B, 1);
vec4 N = mul(K, C);
```

3.12 Tile Management

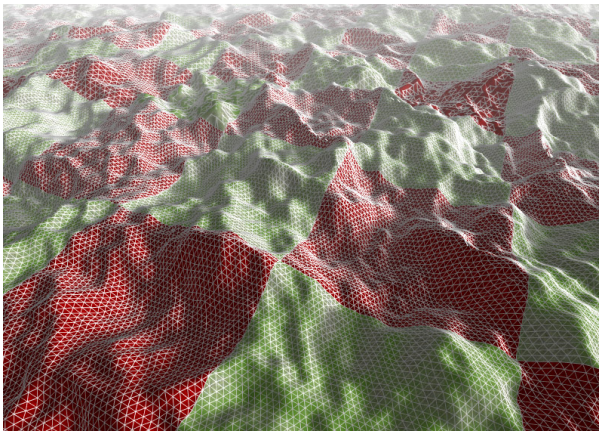


Figure 12: We grid our heightfields into tiles for data management. The tiles colored alternately in red and green are shown.

It is common practice to subdivide large heightfield datasets into a grid of tiles when the entire dataset is too large to hold in video memory (or even main memory!) Although we use an isometric-grid for elevations, we use square tiles to more easily interact with existing LOD algorithms, object management, and of texture mapping.

There are a few files that describe global heightfield parameters: number of tile rows and columns, extent, texture names, and elevation scaling factor. Each tile is stored separately on disk as a compressed binary file. We use the Deflate compression algorithm, which is a combination of LZ77 and Huffman coding implemented in zlib [Deutsch 1996]. This gives an additional compression factor of about 3:1 compression on top of our 6:1 compression for vertex packing(section 3.2). A wavelet pyramid scheme could be easily added to give better compression for relatively low frequency heightfields.

Given a world-space point $P = (P_x, P_y, P_z)$, we find the index of tile under P.

$$P' = \sigma^{-1} P, \tag{12}$$

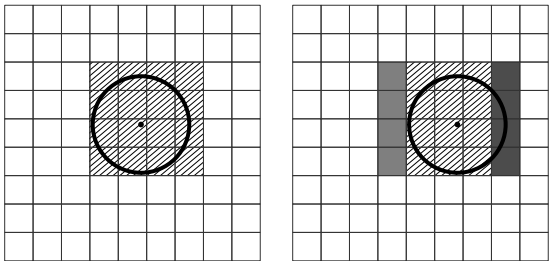
$$t_R = \left\lfloor \frac{P'_z}{extent} \right\rfloor \tag{13}$$

$$t_C = \left\lfloor \frac{P'_x}{extent} \right\rfloor \tag{14}$$

We use a flat array of tiles thus our final index is:

$$tileIndex = t_C + t_R * tileCols\,. \tag{15}$$

Where σ is the object to world scaling matrix, and the heightfield is divided into $tileRows \times tileCols$ tiles.



(a) (b)

Figure 13: A 9x9 grid of tiles, the center point denotes the camera position. (a)The hatched tiles are tiles that are loaded. (b) The camera moves eastward the dark gray column of tiles have to be swapped out and the light gray column are swapped in.

We keep in main memory only those tiles that fall within the bounding box of the circle whose center is the camera's xz position and radius is the distance to the far clipping plane. As the camera moves, rows and columns of this *tile box* are swapped in and out as the box slides over the heightfield. This is illustrated in Fig. 13a, the center point is the camera position, the shaded tiles are the tiles kept in memory.

The implementation is straightforward. We maintain an array of $tileRows \times tileCols$ pointers to our tile objects. We also compute and store the indices of the top and bottom rows, and leftmost and rightmost columns of the tile box. Initially, we create tile objects for all the tiles in the tile box. Each of those tile objects allocates video memory and uploads a portion of the heightfield. While the camera moves during runtime, we translate the tile box. For instance, in Fig. 13b, the camera moves eastward so we need to swap out the light gray column of tiles and swap in the dark gray column. We do this by exchanging the pointers stored in the light gray and dark gray columns, then reinitializing the tile objects pointed to by the dark gray columns with new heightfield data.

There are two classes of boundary conditions. If the initial camera position is close to the edge of the heightfield part of the tile box will be outside of the heightfield. There are two solution to this type of boundary case. Translate the tile box so it is back within the heightfield, or only load the valid portion of the tile box. We chose the latter, and allocate the rows and columns as the tile box moves back over the heightfield. The second type of boundary case requires checking to see that the tile box is never translated off the heightfield.

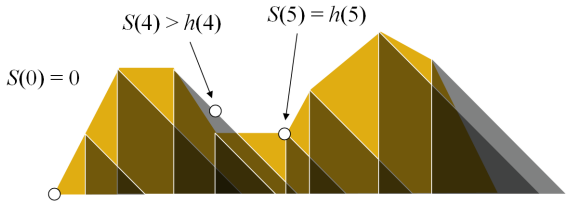


Figure 14: The shadow volumes cast by successive locations moving from left to right enclose one another.

3.13 Directional Light Shadows

The drama of sunrise and sunset and massive scale of mountains is revealed through shadows. Terrain should cast shadows on itself, creating pools of darkness behind large features and distinguishing morning crater bottoms from evening mesas when flying overhead. Terrain should also shadow objects the surface so they do not appear unnaturally bright compared to their surroundings. Terrain should also shadow objects *above* the surface to help disambiguate their elevation. Although these observations are for realistic terrain, they also hold for arbitrary 2D function visualizations that owe their descriptive power to their similarity to geological features.

For any continuous elevation function $h(x, z)$, there exists another function S such that point (x, y, z) is in shadow from h iff $y < S(x, z)$; this follows from the geometry of h. Shadow determination on any heightfield (iso- or ortho-) is thus equivalent to computing S from h. In practice, computing S precisely at mesh vertices and interpolating values between them gives an efficient shadowing solution with the appearance of penumbrae. Linear interpolation artifacts in these fake penumbrae can be mitigated by defining the fractional intensity of the source due to partial occlusion at (x, y, z) by:

$$I = \min(1, \max(0, \left[y - S(x, z) + \frac{k}{2} \right] / k)) \qquad (16)$$

where k is the apparent size of the light source at the shadow caster's distance in elevation units (2.0 is a reasonable value for the Sun and terrestrial mountains).

Assume the light is a single source at 3D homogeneous location $L = (L_x, L_y, 0, 0) = (\cos\theta, \sin\theta, 0, 0)$, e.g., at infinity and along the x-axis, suitable for simulating simulating the Sun or Moon. A point may now only be shadowed by locations on h with the same z value and one of lower or higher x value, depending on the sign of L_x. For simplicity of discussion, let $L_x < 0$ (i.e., "after noon") so that we can work from left to right; $L_x > 0$ can be handled by opposite traversal, and when $L_x = 0$ there are no shadows on a heightfield.

The shadow volume geometry cast by a location $h(x_0)$ (we are now constrained to a single z value) is a right triangle with hypotenuse raised above the horizontal by θ. If the other points on h were at negative infinity, we would have $S(x) = h(x_0) - (x - x_0)\frac{L_y}{L_x}$ for $x > x_0$.

As shown in Fig. 14, the shadow volumes cast by successive locations moving from left to right enclose one another. At each vertex location the new shadow volume is either completely enclosed by the old, making the new redundant, or completely encloses the old one making the old one redundant. Thus we need only a single scalar to incrementally compute S along a row of constant z, and can use simple min-max operations to determine which volume encloses the other. The inductive algorithm for vertices at integer locations is:

$$S(0) = 0 \qquad (17)$$

$$S(x) = \max(S(x-1) - \frac{L_y}{L_x}, h(x)) \quad . \qquad (18)$$

In our method, vertices are stored in row-major format and must be copied to graphics memory in blocks before rendering. We insert the above computation into the copying loop and choose the iteration direction based on the sign of L_x. We compute a self-shadowing value using Equation (16) and pack it into the alpha channel of the per-vertex color attribute. Because the copying process is already dominated by the cost of memory access, the self shadowing term adds negligible time cost. In order to cast shadows onto objects from the terrain, we store the per-vertex S values on the CPU. These are used as the initial S values for the next tile to the right (or left, for opposite iteration).

We compute shadows for only a small number of tiles each frame. Tiles that may shadow visible ones are given highest priority[3]. We also only recompute shadowing when the light source has moved a significant distance.

When the source is very near the horizon, shadows have almost infinite length and the number of non-visible tiles that must be considered grows without bound. This is a drawback we share with other methods.

4 Applications

Figure 15: 250 animated models on Wake Island rendered in real-time.

Isometric heightfields can be used for any application standard ortho-heightfields might be used. We implemented an iso-heightfield terrain rendering engine with the various support algorithms, and applied it to a variety of datasets. Fig. 15 is shot of Wake Island from the game Battlefield 1942 used with permission from Digital Illusions. To simulate the

[3]Most non-visible tiles aren't even in memory and will only have shadowing computed when they are loaded, thus our method's runtime is linear in the number of visible vertices, plus a small constant to account for the just-off-the-screen vertices that may shadow them.

demands of a actual game we render 250 animated models at 28 FPS in addition to the iso-heightfield.

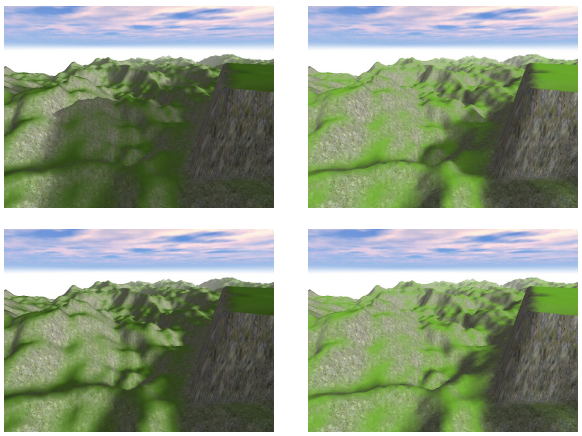


Figure 16: (Top row) Self shadowing terrain; (top left) 6:25am (top right) 9:00am. (bottom row)Terrain with out self shadowing. Notice the moving shadow to the left of the plateau.

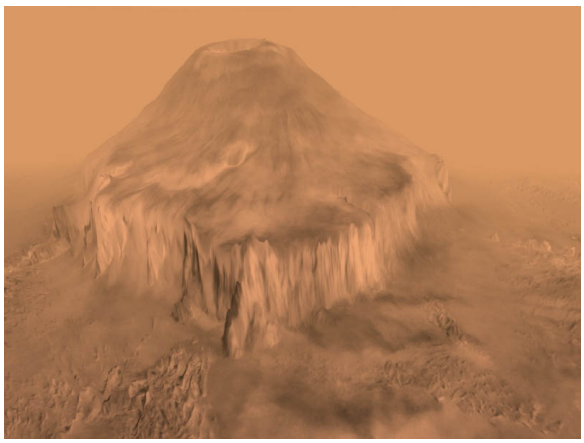


Figure 17: Olympus Mons, 133° 18° Mars. The peak height is about 25 km from the base, three times the height of Mt. Everest.

Scientific visualization is another application of terrain rendering. It is crucial that visualizations minimize artifacts, and err in a predictable manner. A visualization fails when it conveys the perception of features in the data that do not really exist. The reduced error and symmetric error characteristics of the iso-heightfield are thus very attractive for visualization.

We have applied our data structure to a 130° by 30° area centered around the equator of the planet Mars and corresponding registered 32768×16384 satellite photo-montage for a texture. The elevation data consists of 16384×8192 elevation points sampled on a regular ortho-grid. We resampled the data over an isometric grid using bilinear interpolation.

Fig. 17 shows Olympus Mons. Geologists interested how Olympus Mons formed study the cliff and field of ridges at the base called the Aureole. These ridges are thought to be formed from landslides or glacial activity. Notice these small ridges contain high frequency detail that may not be correctly reconstructed from an ortho-heightfield depending on the orientation of feature relative to the grid.

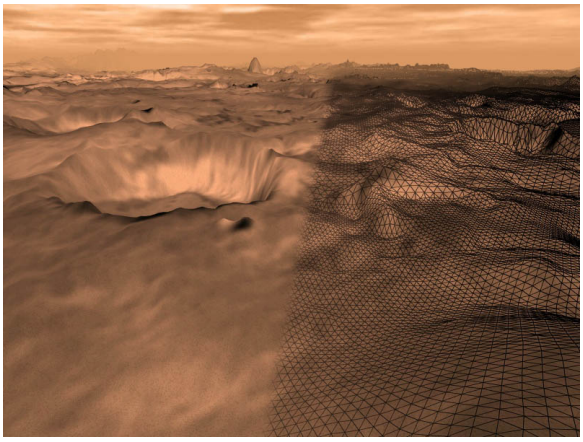


Figure 18: The right half shows the underlying triangle strips used for rendering the surface of Mars. Our terrain rendering application renders the Mars dataset a 130° by 30° area with 250M polygons, in real time.

5 Conclusions and Future Work

This paper adapts several important heightfield algorithms to the iso-heightfield. No doubt there exist equations that are substantially more efficient to implement on the ortho-heightfield, and for which our mesh structure is inappropriate. It is proposed as a complement to the use of ortho-heightfields and not a replacement.

We believe the iso-heightfield is particularly well suited to rendering dynamic surfaces like water waves, the sand beneath player's feet in a volleyball game, and terrain deforming under erosion. It is an interesting open question whether popular mesh animation algorithms like Navier-Stokes and cloth simulation can be effectively adapted to the iso-heightfield.

Acknowledgments

Max McGuire at Iron Lore suggested using an equilateral tiling for real-time heightfields. Hari Khalsa implemented the experiments. This work was advised by John Hughes and David Laidlaw. Morgan's research is supported by an NVIDIA Fellowship. Digital Illusions, CE. granted permission for the Wake Island data. The geology observations are by Dr. James Head III of the Brown University Planetary Geology Department, who provided the NASA Mars data.

References

Akenine-Moller, T., and Haines, E. 2002. *Real-Time Rendering (2nd Edition)*. A K Peters Ltd, July.

Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., and Scopigno, R. 2003. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum 22*, 3 (September), 505–514.

de Boer, W. H., 2000. Fast terrian rendering using geometrical mipmapping. http://www.flipcode.com/tutoirals/geomipmaps.pdf.

Deutsch, L., 1996. Zlib compressed data format specification version 3.

Duchaineau, M. A., Wolinsky, M., Sigeti, D. E., Miller, M. C., Aldrich, C., and Mineev-Weinstein, M. B. 1997. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization Conference Proc.*, 81–88.

Dudgeon, D. E., and Mersereau, R. M. 1990. *Multidimensional Digital Signal Processing*. Prentice Hall Professional Technical Reference.

Harris, M., 2004, August. Remark at NVIDIA U Conference, Los Angeles, CA.

Hoppe, H. 1998. Smooth View-Dependant Level-of-Detail Control and its Application to Terrain Rendering. In *Proc. Visualization '98*, IEEE, 35–42.

Kozlov, S. 2004. Perspective shadow maps: Care and feeding. In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, R. Fernando, Ed. Addison-Wesley.

Levenberg, J. 2002. Fast view-dependent level-of-detail rendering using cached geometry. In *Proc. of the conference on Visualization '02*, IEEE Computer Society.

Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N., and Turner, G. A. 1996. Real-time, continuous level of detail rendering of height fields. In *Proc. of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 109–118.

Losasso, F., and Hoppe, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph. 23*, 3, 769–776.

Max, N. 1999. Weights for computing vertex normals from face normals. *jgt 4*, 2, 1–5.

McGuire, A., 2004, September. Personal Communication, Iron Lore Entertainment, Maynard, MA.

McGuire, M. 2004. Effective shadow volume rendering. In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, R. Fernando, Ed. Addison-Wesley.

Middleton, L., and Sivaswamy, J. 2002. A framework for practical hexagonal-image processing. *Journal of Electronic Imaging, 11*, 104–114.

Middleton, L. 2002. Markov random fields for square hexagonal textures. In *Proc. of 7th International Conference Automation Robotics and Vision(ICARV 02)*.

Musgrave, F. K. 1998. Grid tracing: Fast ray tracing for height fields. Tech. Rep. YALEU/DCS/RR-639, Yale University Dept. of Computer Science, July.

Neophytou, N., and Mueller, K. 2002. Space-time points: 4d splatting on efficient grids. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, IEEE Press, Piscataway, NJ, USA, 97–106.

Pajarola, R. 1998. Large scale terrain visualization using the restricted quadtree triangulation. In *VIS '98: Proceedings of the conference on Visualization '98*, IEEE Computer Society Press, Los Alamitos, CA, USA, 19–26.

Rost, R. J., Kessenich, J. M., Lichtenbelt, B., and Olano, M. 2004. *Open GL Shading Language*. Addison-Wesley Professional, Feb.

Shankel, J. 2002. Fast heightfield normal calculation. *in Game Programming Gems 3, Charles River Media, Inc.*, 344–348.

Theussl, T., Moller, T., and Groller, M. E. 2001. Optimal regular volume sampling. In *VIS '01: Proceedings of the conference on Visualization '01*, IEEE Computer Society, Washington, DC, USA, 91–98.

Thurmer, G., and Wuthrich, C. 1998. Computing vertex normals from polygonal facets. *jgt 3*, 1, 43–46.

Appendix

The following C++ code initializes an index array. We print it here in its entirety because this is the key piece of implementation code and the most likely place for implementors to introduce off-by-one errors. The colors correspond to Fig. 6 where red is an even row, blue is an odd row, and green are degenerate triangles.

```
#define addIndex(_c, _r) \
   {index[i] = _c + (_r) * C; ++i;}
#define addIndex2(_c, _r) \
   {index[i + 1] = index[i] = _c + (_r) * C; i += 2;}

for (int lambda = 0; lambda < LAMBDA; ++lambda) {
  if (lambda >= 1) {                    // STITCH LIST
    std::vector<int>& index = stitchList;
    int i = stitchList.size();

    int numAdded = (3 * (R + C - 7)) / pow(2, lambda - 1);
    int N = index.size() + numAdded;
    detailArray[lambda].numListIndices = N;
    index.resize(N);

    int shift = (lambda == 1) ? 1 : 0;
    int halfstride = pow(2, lambda - 1);
    int stride = pow(2, lambda);

    for (int r = 1; r < R - 2; r += stride) {
        //West
        addIndex(1, r);
        addIndex(1, r + halfstride);
        addIndex(1, r + stride);

        //East
        addIndex(C - 3, r);
        addIndex(C - 3, r + stride);
        addIndex(C - 3 + shift, r + halfstride);
    }

    for (int c = 1; c < C - 3; c += stride) {
      //North
      addIndex(c, 1);
      addIndex(c + stride, 1);
      addIndex(c + halfstride, 1);

      //South
      addIndex(c, R - 2);
      addIndex(c + halfstride, R - 2);
      addIndex(c + stride, R - 2);
    }
  }
  {                                     // STRIP
    int stride = pow(2, lambda);
    int shift = stride / 2;
    std::vector<int>& index = stripArray[lambda];

    int M_strip = ((R - 3) / (stride * 2)) *
          ((C - 4) * 4 / stride + 10) - 3;
    index.resize(M_strip);

    int colEnd = (lambda == 0) ? C - 2 : C - 3;
    int r = 1, i = 0;

    while (r <= R - 4) {
      addIndex(colEnd, r + stride);
      for (int c = C - 3; c >= 1; c -= stride) {
        addIndex(c, r);
        if (c == 1) addIndex(c, r + stride);
              else addIndex(c - shift, r + stride);
      }

      r += stride;
      addIndex2(1, r);
      for (int c = 1; c <= C - 3; c += stride) {
        addIndex(c, r + stride);

        if (lambda == 0) addIndex(c + 1, r);
        else if (c == C - 3) addIndex(c, r);
             else addIndex(c + shift, r);
      }
      r += stride;
      if (r <= R - 4) {
        addIndex(colEnd, r - stride);
        addIndex2(colEnd, r + stride);
      }
}}}
```