Planetary-Scale Terrain Composition

Robert Kooima, *Member*, *IEEE*, Jason Leigh, Andrew Johnson, *Member*, *IEEE*, Doug Roberts, Mark SubbaRao, and Thomas A. DeFanti

Abstract—Many interrelated planetary height map and surface image map data sets exist, and more data are collected each day. Broad communities of scientists require tools to compose these data interactively and explore them via real-time visualization. While related, these data sets are often unregistered with one another, having different projection, resolution, format, and type. We present a GPU-centric approach to the real-time composition and display of unregistered-but-related planetary-scale data. This approach employs a GPGPU process to tessellate spherical height fields. It uses a render-to-vertex-buffer technique to operate upon polygonal surface meshes in image space, allowing geometry processes to be expressed in terms of image processing. With height and surface map data processing unified in this fashion, a number of powerful composition operations may be uniformly applied to both. Examples include adaptation to nonuniform sampling due to projection, seamless blending of data of disparate resolution or transformation regardless of boundary, and the smooth interpolation of levels of detail in both geometry and imagery. Issues of scalability and precision are addressed, giving out-of-core access to giga-pixel data sources, and correct rendering at scales approaching one meter.

Index Terms—Terrain visualization, GPU, GPGPU, render-to-vertex-buffer, level-of-detail.

1 Introduction

The field of real-time terrain rendering research is vibrant and has been so for decades. Motivations dating back to the birth of computer graphics focus on flight simulation as the driving application for real-time algorithms, and since then, interactive visualization has been embraced by broad communities of scientists including geologists, seismologists, climatologists, and planetary scientists. Real-time terrain visualization is a fundamental tool in the search for oil on Earth, water on Mars, landing zones on the Moon, and countless other areas of exploration.

A vast quantity of data exists describing the Earth and other planets, with height maps encoding their terrain and features, and surface maps encoding a variety of quantities including color in many wavelengths. The presentation, projection, resolution, and coverage of these data sets are appropriate to their subject, and thus, are as varied as the data they represent. But, all of the data relating to given planet *are* related and there is a clear motivation to bring these data together in a common visualization.

The challenge lies in merging them correctly and efficiently. It does not suffice merely to draw one height field after another. Depth buffering causes overlapping surfaces to obscure one another in unpredictable ways. We must instead compose height maps before the geometry is drawn.

Neither is it straightforward to merge surface maps atop rendered height geometry, as these maps may be unregistered with the height map underlying them, and each application of additional textures incurs the expense of rerendering of some or all geometry. There is, thus, a motivation to decouple surface texture application from geometry entirely.

Traditional approaches entail the merging of disparate data sources in a preprocessing phase, which resolves any mismatches in projection and resolution. However, we cannot preprocess all source data to a single common projection, as no single projection can optimally represent all data at all points on the globe. Similarly, we cannot preprocess all source data to a single resolution, as this would require an expansion of global low-resolution data to accommodate the inclusion of local high-resolution data. To allow for both optimal projection and optimal resolution globally, we must instead approach the composition of terrain data in a real-time, interactive, view-adaptive fashion. In so doing, we preserve the identity and quality of the original data.

Given the capability of modern graphics processors to both read from and write to 4-channel 32-bit IEEE floating-point frame buffers, the practice of GPGPU programming has emerged. This practice views the GPU not only as 3D renderer but also as a parallel vector processor. Related extensions to OpenGL have enabled the ability to apply a buffer of color values in the context of a buffer of geometry values, blurring the distinction between colors (r, g, b, a) and vectors (x, y, z, w).

These advances allow us to process vertices as pixels, and thus, formulate geometry processes in terms of image processing. A unification of terrain height map and surface map processing follows from this. The alpha-compositing of

- *R. Kooima, J. Leigh, and A. Johnson are with the Department of Computer Science (MC 152), Electronic Visualization Laboratory, University of Illinois at Chicago, Room 1120 SEO, 851 S. Morgan St., Chicago, IL 60607-7053. E-mail: rlk@evl.uic.edu, spiff@uic.edu, lenzman@mac.com.*
- *D. Roberts and M. SubbaRao are with the Adler Planetarium and Astronomy Museum, 1300 S. Lake Shore Drive, Chicago, IL 60605. E-mail: {droberts, msubbarao}@adlerplanetarium.org.*
- *T.A. DeFanti is with the California Institute for Telecommunications and Information Technology (Calit2), University of California, San Diego, Atkinson Hall, 9500 Gilman Drive, La Jolla, CA 92093-0436. E-mail: tdefanti@soe.ucsd.edu.*

Manuscript received 8 Sept. 2008; revised 30 Dec. 2008; accepted 8 Apr. 2009; published online 17 Apr. 2009.
Recommended for acceptance by M. Chen, C. Hansen, and A. Pang.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-2008-09-0140.
Digital Object Identifier no. 10.1109/TVCG.2009.43.

imagery has been common throughout the history of real-time 3D graphics, but with the distinction between geometry and imagery blurred, the same compositing capability becomes possible with geometry. We gain the ability to manipulate and compose surface map data on the fly and height map geometry as well.

Given this, we seek to create tools for highly scalable planetary rendering and data composition using little-to-no preprocessing. We require only that large data sets be tiled and mipmapped, a capability provided by off-the-shelf tools such as GlobalMapper [1].

1.1 Related Work

Real-time terrain rendering algorithms have evolved along side 3D graphics hardware. Early approaches such as Duchaineau et al.'s ROAM algorithm [10] and Lindstrom et al.'s continuous level-of-detail method [18], [17], [19] devoted significant CPU resources to selecting an optimal set of triangles to represent a height field. Such approaches reflected the limited geometry processing capacity of the 3D rendering hardware of the day.

The arrival of powerful GPUs shifted the focus away from CPU triangle processing and toward bulk GPU rendering of large blocks of geometry. de Boer's geomipmapping algorithm [8] is a straightforward approach using a quad-tree of geometry grids. Levenburg [16] provides a more powerful mechanism, improving upon ROAM to produce cacheable static binary trees of triangles. Pajarola and Gobbetti [24] provide a detailed survey of similar work, up to and including techniques for geometry block rendering.

As GPU flexibility increased, per-frame on-the-fly manipulation of terrain data came to the fore. Schneider and Westermann [28] exploit vertex shading to interpolate between blocks of geometry, achieving continuous level-of-detail. Losasso and Hoppe's geometry clipmap approach [21] utilizes an innovative toroidal data update, allowing concentric blocks of geometry to remain centered upon the viewer.

The treatment of geometry as imagery appeared in the work of Gu et al. [13], enabling arbitrary 3D meshes to be operated upon by image processing algorithms, notably image compression techniques. Dachsbacher and Stamminger [7] would later apply this method to terrain maps, giving an elegant method of producing view-adaptive terrain geometry in a GPU-friendly fashion, and paving the way for basic terrain composition in the form of procedural detail generation.

As terrain scale increases, terrain data sets may no longer be assumed flat. A number of high-efficiency planetary-scale spherical terrain rendering approaches have been proposed to address this. In particular, Clasen and Hege adapt the geometry clipmap concept to the sphere [6] giving an advanced GPU-centric technique.

Cignoni et al. present Planet-sized Batched Dynamic Adaptive Meshes (P-BDAMs) [5], a technique for rendering a hierarchy of right-triangulated irregular networks with associated texture maps. This approach entails extensive preprocessing of the terrain mesh and textures. In contrast, we seek to move the triangulation process to the GPU, working directly from the source height maps, with the terrain mesh untouched by the CPU.

Most terrain visualization techniques are concerned primarily with rendering terrain geometry efficiently, with little attention paid to issues of flexible data visualization. Döllner et al. address this gap [9]. They present a model for large-scale texture paging and propose texture mapping as a fundamental primitive in terrain visualization, giving a number of unique surface data layering tools ("lens" masking, animation, and topographic representation) defined in terms of multipass texture application. Our work further generalizes this type of real-time surface map composition by extending the capability to the composition of underlying height maps.

Bruneton and Neyret [4] touch upon height map composition, providing an approach to the real-time generation of both the geometry and appearance of terrain. Beginning with an elevation map and a vector definition of surface features, they use GPU fragment processing to generate and cache the diffuse color of the landscape and carve vector features such as lakes, rivers, and roads into the elevation map itself. Our work is similar in spirit, omitting the composition of vector data, but allowing the composition of multiple elevation maps, and mapping each onto a geometric structure more appropriate to the sphere.

GPU-based geometry processing is certainly not limited to terrain rendering. Boubekeur and Schlick present a generic technique for static [2] and later adaptive [3] mesh refinement wherein a GPU vertex shader maps refined triangular meshes onto coarse CPU-specified triangular patches. Height field displacement mapping follows directly from this, but the application of this technique to spherical terrains is limited. The arc length of a spherical triangle's edges varies with the size of the triangle, so interpolation using barycentric coordinates results in inconsistent refined edge lengths at large scales. This limitation motives our use of an iterative midpoint subdivision technique, which gives consistent edge lengths at all scales.

1.2 This Discussion

Section 2 defines our method in detail, laying out the process by which height map and surface map data are displayed, and pointing out opportunities for on-the-fly manipulation and composition. Section 3 describes ways of exploiting these opportunities to height and surface data, enabling automatic adaptation to nonuniform data sampling, the seamless blending and overlay of unregistered data, and straightforward application of out-of-core paging and level-of-detail interpolation.

In an effort to make the discussion of the effects of our method concrete and clear, this presentation consistently employs the false data example planet shown in Fig. 1. All of these figures were rendered using our implementation of the approach. The false data exaggerate the issues of sampling and scale that this work focuses upon. This planet was generated using 3D simplex noise [25], giving a magnified height field. As an entity independent of data sampling, the simplex noise planet was "observed" using a variety of projections and resolutions in order to model the variance among data sets of real planets. A color map with a high-contrast, low-resolution contour line was generated from the sampled height maps and is drawn *without* linear

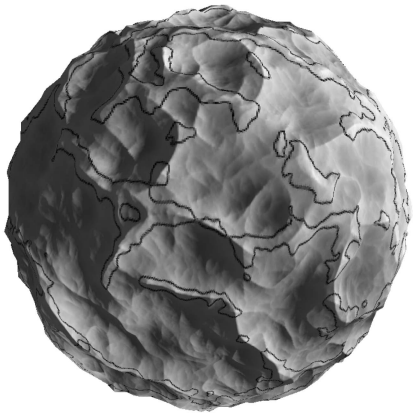


Fig. 1. The 3D simplex noise planetary height map with normal and diffuse color maps, used for subsequent figures.

filtering so that nonuniform data sampling is made apparent in the shape of the rendered texels. The application of this approach to real large-scale data is demonstrated in the performance analysis in Section 4.

2 Method

The method proceeds in three phases, shown in Fig. 2: CPU visibility and level-of-detail determination, GPU geometry generation, and GPU rendering and illumination. The visibility phase determines a coarse set of visible triangular surface patches, the geometry generation phase refines these to give a displaced terrain mesh, and the rendering phase rasterizes the mesh and applies surface textures. This two-step geometry refinement process is similar to the work of Lindstrom et al. [18], [17], [19], though our refinement process is offloaded to the GPU.

The three phases are distinguished from one another by the explicit transfer of the output of one to the input of the next. When rendering multiple planetary bodies, these phases are interleaved, exploiting the parallelism of processing and data transfer.

2.1 Visibility and Granularity

The visibility and granularity phase is executed by the CPU. The primary goal of this phase is to produce a coarse triangulation of the visible portion of the sphere, refined to give consistent level-of-detail. This subdivision is *not* the final spherical tessellation, it is merely a gross determination of visibility and granularity. To produce a uniformly triangulated sphere, we begin with the icosahedron and proceed by recursive subdivision of its faces, as in Fig. 3.

The use of the icosahedron as base polyhedron is not strictly necessary, but we select it for the uniformity of its triangulation. A variety of polyhedral tessellations have been used for planetary-scale rendering. For example, the spherical-ROAM texture tiling method by Hwa et al. [14] uses a cube, as cubic subdivision retains the right triangular tessellation used by their 45-degree tile rotation.

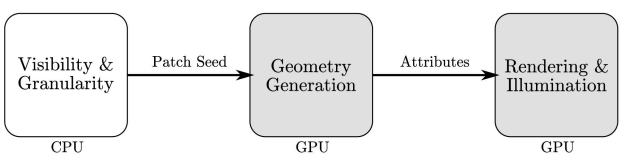


Fig. 2. The three phases of the planet rendering pipeline.

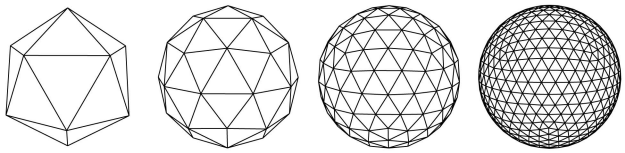


Fig. 3. Recursive subdivision of the icosahedron gives a sphere of very uniform triangulation.

Height map data will be used to displace our refined geometry in subsequent rendering passes and these height maps are expected to represent a variety of projections and resolutions. We do not favor one projection over another, so we resample all incoming data to the icosahedron, as it is the geometric form that best minimizes geometry tessellation artifacts.

Throughout the runtime, we maintain a frame-coherent hierarchy giving the current faces of the icosahedral subdivision. This is a tree structure with a constant number of leaves. The management of this tree proceeds similarly to the ROAM algorithm. But while ROAM is concerned with the maintenance of a continuous triangulation free of T-intersections, we are concerned only with a coarse triangulation. Thus, we have the luxury of ignoring T-intersections until after refinement (Section 2.3) and our hierarchy maintenance algorithm is simplified accordingly.

As the view varies from frame to frame, those triangles that move into the view frustum are added to the tree, and those that move out of the view are pruned. We seek a set of visible triangles numbering as near as possible to (though not larger than) a constant n_t and maintain this set with a simple priority algorithm. Triangles are sorted by the solid angle that each subtends from the current view. If the set of triangles is too large, the smallest set of four sibling triangles is collapsed. If the set of triangles is too small, the largest triangle is subdivided. The result is a coarse set of approximately n_t visible triangles. These appear as in Fig. 6a.

Testing these triangles for visibility is more complex than simply checking each against a view frustum, since the surface of the planet within the triangle's bounds is not planar. Height data will be mapped within that area, and we must determine whether the geometry of this terrain is or is not visible. As shown in Fig. 4, the three sides of a triangular area on a sphere are segments of geodesics, or "great circles" cutting the sphere into equal halves. These geodesics define three planes cutting through the center of the sphere, forming a wedge-shaped volume. The terrain within this volume has some minimum and maximum altitude, which may be conservatively assumed to be the minimum and maximum radii of the planet. The three planes and two radii define a surface *shell*, a tight bound on the terrain within the triangle. We may determine the visibility of the terrain within the triangle by testing the triangular bounding shell.

Having adapted our triangulation of the icosahedron to the current view, the three vertices of each of the n_t triangles are stored in a trio of $3 \times n_t$ textures, each of which is as

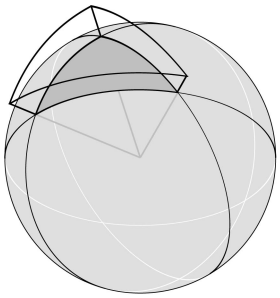


Fig. 4. A triangular surface shell, a terrain bounding volume defined by three planes and two radii.

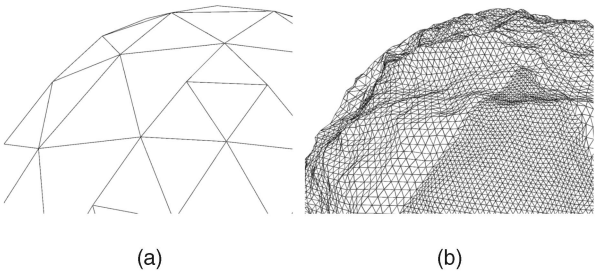


Fig. 6. A triangulation of the visible portion of the sphere, as produced by the visibility and granularity phase (a) and refined by the geometry generation phase (b). (a) Visibility output. (b) Geometry output.

depicted in Fig. 5. One texture gives the eye-space vertex position, another gives the eye-space vector normal to the sphere, and the third gives the spherical coordinate (longitude and latitude). In the following GPU-based phase of the process, these will be used to determine coordinates for height map texture reference. Height values will be used to displace the vertex position along the normal, giving eye-space terrain geometry. The three vector values are uploaded to GPU-local storage asynchronously using a pixel buffer object. This is the initial point at which geometry is represented as imagery and this representation will persist throughout the rendering process.

This is also the first point at which issues of precision are addressed. To accommodate meter scale data in a planetary-scale context, the coarse triangulation is computed in double precision floating point on the CPU and transformed into eye space before being uploaded to the GPU. Eye space is a coordinate system centered upon the viewer rather than the planet. This allows the fine granularity of small floating-point values to be useful near the viewer and the coarse granularity of large values to be pushed off to the distance. This necessitates the resubdivision of the sphere each time the viewpoint moves, but the cost of it is negligible in practice, as will be demonstrated in the performance analysis in Section 4.

This use of CPU-based double precision and manipulation of coordinate systems is common to a number of prior approaches. Reddy et al.'s TerraVision II [26] and Lindstrom et al.'s VGIS [17] position data within local coordinates, manipulating the model-view matrix as each data subset is rendered. Given our goal of compositing all geometry in a common coordinate system *before* rendering, we do not have the luxury of varying the transformation across multiple geometry render passes, so the direct specification of eye-space coordinates follows.

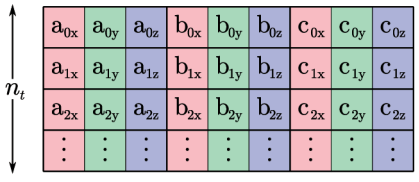


Fig. 5. The geometry generation seed of n_t triangles, each with three vertices $[a, b, c]$, encoded as a $3 \times n_t$ RGB texture. There are three such: position, normal, and spherical coordinates.

Cignoni et al.'s P-BDAM [5] uses eye-space coordinates in a similar and especially elegant fashion. P-BDAM specifies the internal vertices of each triangular geometry patch in terms of barycentric coordinates, relative to the three outer vertices, which are supplied as vertex shader uniforms. Like our approach, this does still entail the recomputation of each eye-space vertex position per frame. While P-BDAM performs this task in the vertex shader, we do in the fragment shader.

2.2 Geometry Generation

The goal of the geometry generation phase is to produce a high-resolution triangular tessellation of the sphere, accurately representing the terrain of the input height maps, with consistent level of detail, as shown in Fig. 6b. In this process, the coarse eye-space triangulation provided by the CPU is further refined by the GPU using recursive subdivision.

A triangle subdivided to depth d has $n_v(d)$ vertices:

$$n_v(d) = \frac{(2^d + 1)(2^d + 2)}{2}.$$

With the exception of the three initial vertices defining a triangle (v_o, v_1, v_2), every vertex v_i is the combination of two other vertices v_j and v_k. A *breadth-first* order enumeration of the n_v vertices has the property that $i > j$ and $i > k$ for any such related vertices v_i, v_j, and v_k. Fig. 7 depicts this relationship graphically, with arrows indicating dependence.

2.2.1 The GPGPU Approach

The geometry subdivision phase is a GPGPU process that "ping-pongs" a pair of $n_v(d) \times n_t$ render targets, processing all triangular surface patches in parallel, and iteratively computing one new level of subdivision with each step. See Fig. 8. At each step s, a rectangle is drawn from $[n_v(s), 0]$ to $[n_v(s+1), n_t]$, triggering the fragment program for each new vertex.

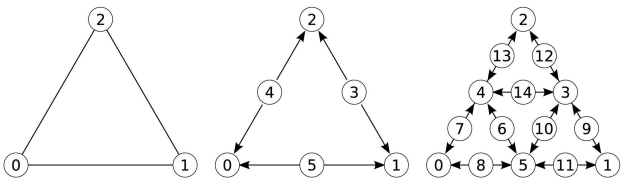


Fig. 7. Subdivided vertex indexes and dependencies, depth $d = 2$.

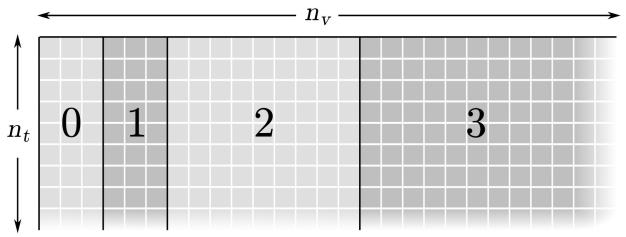


Fig. 8. The GPGPU geometry generation buffer. Each pixel gives a vertex in the final tessellation. The $3 \times n_t$ section 0 corresponds to Fig. 5.

A lookup table (Fig. 9) indicates which of the already-computed vertices must be averaged to give the current vertex. These are referenced from the "ping" texture and the new vertex is written to the "pong." The "ping" and "pong" are logically swapped and the process repeats until the configured depth is achieved, usually five or six steps depending on desired performance/quality. This gives 561 or 2,145 generated vertices per triangle, respectively, far more than is possible using current OpenGL geometry shading capability.

In short, this is a recursive geometric process formulated as an iterative image process. This perpetuates the treatment of geometry as imagery, the concept at the core of this approach. This subdivision is performed for each of the three attribute buffers supplied by the CPU, resulting in fine-grained eye-space positions, normals, and spherical coordinates.

2.2.2 Vertex Normal Computation

Like positions, vertex normals are stored in eye space. These vectors are normal to the smooth sphere, so subdivision may be performed using bisection. The vectors $\bar{n}_j$ and $\bar{n}_k$ are the normals of the input vertices, and $\bar{n}_i$ is the resulting subdivided normal:

$$\bar{n}_i = \frac{\bar{n}_j + \bar{n}_k}{\|\bar{n}_j + \bar{n}_k\|}.$$

Here, a trick is introduced. The geometry render target is a four-channel floating-point frame buffer and the as-yet-unused 4th channel is used to hold the angle separating the input normals. This will be used during position computation.

$$w = acos(\bar{n}_j \cdot \bar{n}_k).$$

2.2.3 Longitude and Latitude Computation

Under ideal circumstances, it would not be necessary to compute the longitude and latitude of each vertex by recursive subdivision. Normally, one would not even bother to store these coordinates, preferring to simply compute them from the vertex normal at render time. However, this is imprecise and unstable. Generating spherical coordinates in this fashion leads to significant quantization, resulting in surface mapping errors on the order of hundreds of meters on the scale of the Earth, with a total failure of interpolation across the International Date Line.

$i=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$j=$	-	-	-	1	2	0	4	4	0	1	5	5	3	2	3	...
$k=$	-	-	-	2	0	1	5	0	5	3	3	1	2	4	4	...

Fig. 9. Vertex dependence index lookup table, encoding the relationship depicted in Fig. 7. Vertex i depends upon vertices j and k.

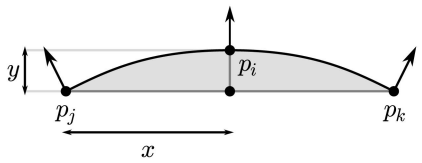


Fig. 10. Computing precise position subdivision without reference to radius. $\bar{p}_j$ and $\bar{p}_k$ are the input vertex positions, and $\bar{p}_i$ is the computed midpoint.

The haversine geodesic midpoint method more reliably subdivides the coordinates of the input vertices. This method computes the midpoint of the shortest path between two positions specified as longitude (θ) and latitude (ϕ). It correctly handles longitudes outside of the range $[-\pi, +\pi]$ without wrapping and latitudes near the poles without loss of longitude. (θ_j, ϕ_j) and (θ_k, ϕ_k) are the coordinates of the input vertices, and (θ_i, ϕ_i) is their computed midpoint. b_x and b_y are temporary variables:

$$\begin{aligned}
b_x &= cos\ \phi_k \cdot cos(\theta_k - \theta_j),\\
b_y &= cos\ \phi_k \cdot sin(\theta_k - \theta_j),\\
\phi_i &= atan2\big(sin\ \phi_j + sin\ \phi_k, \sqrt{(cos\ \phi_j + b_x)^2 + b_y^2}\big),\\
\theta_i &= \theta_j + atan2(b_y, cos\ \phi_j + b_x).
\end{aligned}$$

2.2.4 Vertex Position Computation

Position subdivision is particularly picky, as it is especially prone to numerical imprecision. The common process is to scale the normal by the average of the input radii and offset from the center of the planet (as we are working in eye-space rather than object-space). Unfortunately, multiplication by a large radius value may consume more precision than can be represented by a 32-bit float. Once again, geometry on the scale of a meter becomes dominated by numerical precision artifacts. We need a method that linearizes at small scales and does not make explicit reference to the radius of the planet.

See Fig. 10. We take the angle between the input normal vectors (as found during normal computation) and compute its tangent using a constant lookup table stored as a 1D texture map. We compute x, half the distance between $\bar{p}_j$ and $\bar{p}_k$, and multiply it by the tangent giving y. From there, we offset the midpoint of $\bar{p}_j$ and $\bar{p}_k$ along the current normal $\bar{n}_i$ by the distance y.

As the input normals tend toward equality, the computation of their angle reliably tends toward 0 degree with little noise. Thus, the table lookup reliably tends toward $y = 0$, and the position offset reduces to the midpoint of the input points. Each step preserves its precision and the stability of the operation as a whole actually increases as the scale decreases.

We now have everything we need to begin applying height map data to the tessellated sphere.

2.2.5 Vertex Displacement

The final geometry of the spherical terrain is computed using a *geometry accumulation* buffer. As with the geometry generation buffer, each pixel of the geometry accumulation buffer corresponds to a vertex in the spherical tessellation. When height map textures are blended with this buffer, these vertices are displaced, giving the geometry of the terrain.

In this process, each height map data set is loaded and bound as a texture. The position, normal, and spherical coordinate buffers we have just computed are also bound as textures. A rectangle is drawn to the geometry accumulation buffer, and at each fragment, the spherical coordinate (or the normal, for polar data) gives the texture coordinate used to reference the height map value. The position is offset along the normal by this value and blended with the output.

This accumulation allows a multitude of height maps to be combined to form the final visible geometry. Regardless of projection or resolution, each of these height maps is resampled from its native image space to the 2D logical space of the tessellation. Because of this resampling, discontinuities due to projection, data boundary, and disparities in resolution are eliminated. Texture magnification filtering interpolates low-resolution data linearly and minification filtering down-samples high-resolution data.

Geometry accumulation also affords an opportunity to enhance low-resolution height maps with procedurally generated detail, as demonstrated previously by Losasso and Hoppe [21] and Dachsbacher and Stamminger [7]. While possibly not appropriate for scientific visualization, procedural terrain displacement can add welcome detail to landscapes in game and entertainment applications, at little to no cost in data storage. To accomplish this, we need only blend fractal or Perlin-style noise with the existing geometry accumulation buffer using an appropriate fragment shader.

The per-frame resampling of height map data may appear wasteful of GPU resources, but since it is expressed in terms of common texture mapping operations, it is no different than the per-frame resampling of texture data applied to an ordinary 3D mesh. As such, the overhead of data resampling is minimal, as demonstrated in the performance analysis in Section 4.4.

Our approach to height map accumulation affords an opportunity to satisfy the last criterion for continuity. Lindstrom et al. [18] enumerate three aspects of continuity of terrain level-of-detail under viewpoint motion. In summary, they are 1) geometry morphs between discrete levels of detail instead of popping, 2) adjacent blocks of geometry align without gaps, and 3) the number of triangles tessellating a given area varies smoothly. In our method, 3) is satisfied during the visibility and granularity phase (Section 2.1) and 2) is satisfied during rasterization (Section 2.3), but 1) remains.

However, with our approach, height geometry displacement is expressed in terms of image composition and performed by pixel processing hardware. Thus, morphing between geometric levels of detail is equivalent to blending between images. Just as height map resampling is accomplished by the GPU's bilinear texture filtering capability, so too may level-of-detail continuity be satisfied using the GPU's trilinear mipmapping capability. The mipmap bias is the fractional part of the level-of-detail coefficient, as used in existing approaches to geomorphing [12]. At the time of this writing, this advanced terrain sampling operation is untested and our current implementation does reveal popping artifacts. Recognizing the necessity of continuous LOD in any modern approach to terrain rendering, we place trilinear filtered geometry among our most important areas for future work.

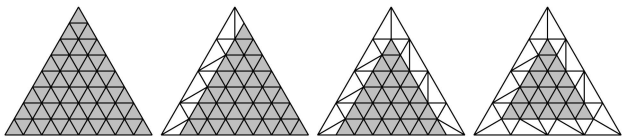


Fig. 11. Vertex windings needed to eliminate T-intersections, adjoining high-granularity geometry with adjacent lower granularity geometry.

After all input height maps have been accumulated, the final vertex attributes are concatenated to a vertex buffer object for rasterization. Given the input shown in Fig. 6a, the output shown in Fig. 6b is generated. Each vertex of the tessellation has a position, normal, and longitude/latitude texture coordinate.

2.2.6 Aliasing and Error

Our height map displacement process does not enforce a constraint that the vertices of the source data fall upon the vertices of our triangulation. The vast majority of sampled height map values are the result of texture filtering, and thus, a linear interpolation of the four surrounding values. This is a significant departure from one of the basic assumptions made in the field of terrain rendering. Filtered terrain texture sampling is not unheard of and Livny et al.'s persistent grid ray-casting approach [20] is one example, but most terrain rendering approaches utilize a regular grid of data applied to a regular geometric grid of right triangles. Our goal of supporting arbitrarily projected data precludes this and the impact is twofold. First, it allows aliasing to occur when data are resampled to our triangular mesh. Second, it adds complexity to the analysis and mitigation of error.

While existing planetary-scale terrain rendering literature does discuss the mapping of spherically projected data onto subdivided polyhedra, little attention has been paid to the effects of the nonuniformity of such mappings on aliasing and error. We understand the critical necessity of a thorough analysis of the error and failure modes that arise here, but we leave this analysis for our future work.

2.3 Rendering and Illumination

The rendering phase begins by determining an appropriate triangular winding of the vertices generated previously. The neighborhood of each coarse triangle determines whether a level-of-detail transition has occurred along its border. Such a transition may result in adjacent lower resolution geometry along zero, one, two, or three sides. An element buffer object is selected accordingly from among those shown in Fig. 11, producing a triangulation of the surface free of T-intersections.

2.3.1 Deferred Texturing

This geometry is drawn to an offscreen four-channel floating-point buffer with fragment color (θ, ϕ, n_x, n_z). That is, only the values of the spherical coordinate and normal at each point are drawn. This *deferred texturing* technique allows a large number of surface map textures to be composed atop terrain geometry without the need to make multiple geometry rendering passes. This is a generalization of deferred shading,

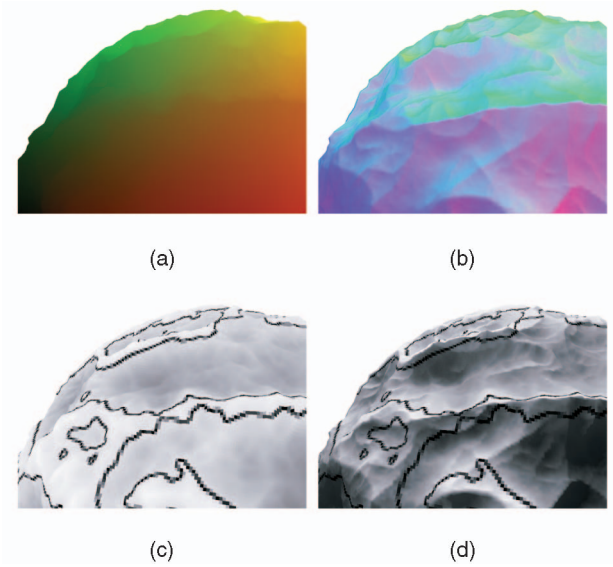


Fig. 12. The accumulation buffers used during the screen-space processing of surface maps. (a) Spherical coordinates, θ in red, ϕ in green. (b) Normal accumulation. (c) Diffuse accumulation. (d) Final image.

a technique Saito and Takahashi developed and referred to as "G-buffers" [27], where offscreen buffers receive the color and normal of rendered geometry, allowing the contribution of multiple light sources to be applied in screen space.

Rendering is $O(n)$ in the number of vertices drawn. If m surface maps are to be applied to this geometry, then traditional multipass rendering is $O(n \cdot m)$. Deferred texturing incurs the cost of geometry rendering at most once and m surface maps may be applied in $O(n+m)$ time. In the analogous case, G-buffering allows m light sources to be applied in $O(n+m)$ time.

Fig. 12a shows the resulting texture coordinate buffer, with longitude (θ) in the red channel and latitude (ϕ) in the green channel. Note that this is the *only* time 3D geometry is drawn and all subsequent rendering is performed in screen space.

2.3.2 Surface Map Accumulation

Just as height maps of arbitrary projection and type are adaptively composed as images during geometry generation, so too are surface maps composed during rendering. With height maps operated upon in the logical space of the tessellation and surface maps operated upon in screen space, we see that all data are processed in the space in which they are *used* rather than the native space in which they are presented.

During surface map accumulation, the bounding volume of the surface data set is drawn. For each fragment, the texture coordinate is retrieved from the deferred texture buffer, with spherical data using (θ, ϕ) and polar data using (n_x, n_z). Additionally, the depth buffer value and projection inverse may be used to compute the eye-space position (p_x, p_y, p_z) of each fragment, which gives means to find coordinates for orthogonally projected or perspective projected surface data, such as photographs. Finally, the surface data texture is referenced. This is another opportunity to perform a data manipulation, such as projection-quality adaptation, level-of-detail blending, or procedural detail generation.

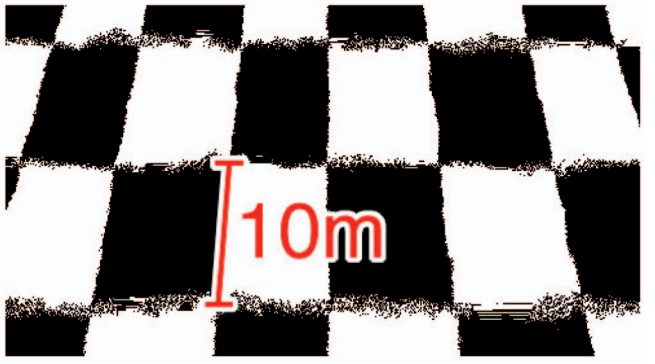


Fig. 13. A procedurally generated color map showing the scale and character of the numerical imprecision of the texture coordinate buffer. Squares are 10 m high, applied to a sphere the size of the Earth (mean radius 6,372,797 m).

Each distinct type of surface map data is composed separately to an offscreen buffer. Fig. 12b shows the accumulated normal maps of our example planet. Fig. 12c shows the accumulated diffuse color maps.

Finally, an onscreen pass is made in the form of a full-screen rectangle. For each fragment, each of the accumulated surface map buffers is referenced, combined as needed, illuminated as desired, postprocessed, and displayed, as shown in Fig. 12d.

2.3.3 Texture Coordinate Buffer Precision

The maximum resolution of a surface texture map renderable with this technique is limited by the precision of the 32-bit IEEE floating-point values stored in the texture coordinate buffer. A 32-bit float provides 24 bits of significand (23 bits plus one implicit lead bit), so the smallest feature uniquely representable using spherical coordinates at any point on a sphere of radius r has size $2\pi r/2^{24}$ meters. On a sphere the size of the Earth, with mean radius 6,372,797 meters, this works out to 2.39 meters. On the moon, with radius 1,738,400 meters, the value is 0.65 meters.

To make the effects of this imprecision apparent, a planet the size of the Earth is rendered using a procedurally generated texture. The texture coordinate (θ, ϕ) is read from the buffer, normalized to $[-1, 1]$, scaled by a factor of $\approx 1{,}001{,}036$, and fed to a step function. This gives a black-and-white checkerboard pattern, where each square has a height of 10 meters on a sphere of radius 6,372,797. The checkerboard width is 10 meters at the equator, narrowing to zero at the poles.

Fig. 13 shows the resulting color buffer as it appears near the international data line and the arctic circle. We can see that the coordinates are usable at a scale of 10 meters, but are largely noise at a scale of one meter. This is the worst-case behavior, with the effect of the noise tending toward zero near the equator and the prime meridian.

We may potentially work around this limitation using a technique similar to that used during geometry generation. There, geometry is generated in viewer-centric eye coordinates to ensure that values remain near zero. Here, we find the texture coordinate of the point on the sphere beneath the viewer and generate texture coordinates offset from that. As before, values near the viewer remain near zero.

Such offset spherical coordinates trigger a cascade of complications throughout the rendering process. In particular, the calculation of the midpoint of two coordinates no

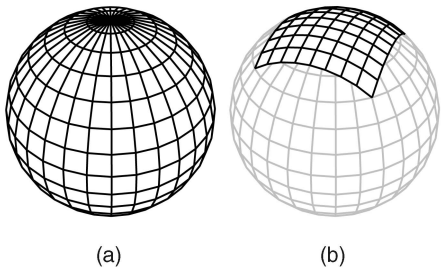


Fig. 14. Common planetary projection types. (a) Spherical projection. (b) Polar projection.

longer follows from the formulation given in Section 2.2.3. We have yet to derive a stable formulation of this part of the process.

Offset coordinates also complicate the correct application of surface textures. It does not suffice to merely apply the texture coordinate offset of the viewer when making the texture reference, as to do so would reintroduce a value far from zero. Instead, the texture itself must be "virtually" shifted. Efficient implementation of this follows from the texture paging index mechanism, which will be described in Section 3.3.

The ultimate solution is the use of double precision floating point. High-end GPUs with double precision capability from both NVIDIA and AMD are on the market in 2008. Double precision emulation has been explored, though the emulation of trigonometric functions is expensive and involved.

3 Real-Time Data Manipulation Results

The true value of this approach to the display of height map and surface map data is that it provides a number of opportunities for on-the-fly data manipulation. Having established that arbitrary juxtaposition, weighting, and blending of both height and surface data are possible in real time, a number of useful results follow immediately. The following sections describe these. In all cases, the displayed figures were rendered by the implementation of our approach.

3.1 Projection Quality Adaptation

Planetary data are most often presented using spherically projected images. The x-axis of the image maps directly onto longitude and the y-axis maps directly onto latitude. Image axes map onto the sphere as in Fig. 14a. The Shuttle Radar Topography Mission (SRTM) [11] data set, a 1-arcsecond height map of the Earth, is a common example of a spherical data set. The Shuttle's orbit limits the extent of this data set to approximately 60 degree above and below the equator and the spherical projection is optimal.

However, consider Blue Marble Next Generation (BMNG), the familiar mosaic of color Earth imagery. It too is spherically projected, but extends all the way to the poles. Spherical projection suffers at the poles, where all lines of longitude, and thus all columns of image data, converge. Pixels become compressed along the x-axis, while retaining their size along y. The visual effect of this is a radial blur centered at the pole. This anisotropic sampling also imposes a significant data access penalty, as large quantities of source data (the entire width of the source image) must be accessed when rendering the pole. This taxes VRAM utilization and may cause data caches to thrash. Bad polar sampling is extremely common in planetary data visualization.

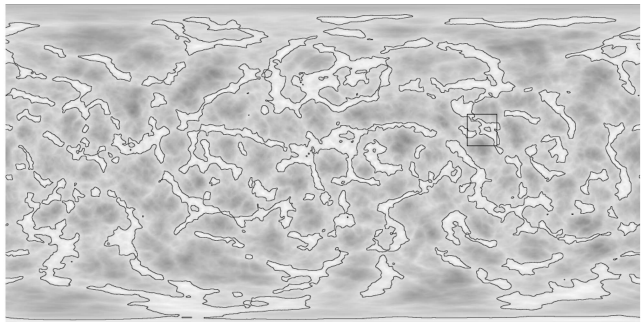


Fig. 15. The spherical projection of the diffuse color of the example planet. The marked region corresponds to Fig. 18.

The correct solution for polar rendering is to use polar-projected source data, as in Fig. 14b, where the sampling of the source most closely matches the sampling of the rendered image. The Landsat Image Mosaic of Antarctica (LIMA) is an example of a high-resolution data set presented with polar projection. In particular, we work closely with geoscientists at the Antarctic Geospatial Information Center (AGIC) at the University of Minnesota. This group has specific need for the means to interactively compose large quantities of high-resolution localized data near Earth's south pole and to visualize these data in the context of Earth as a whole.

To render an entire planet with uniform sampling, both spherical and polar projections are required. Planetary data sets providing *both* of these are rare, but the Mars Orbiter Laser Altimeter (MOLA) data set is one example. It provides spherical projection of Mars height data up to 88 degree from the equator, filling the gap at each pole using data with polar projection.

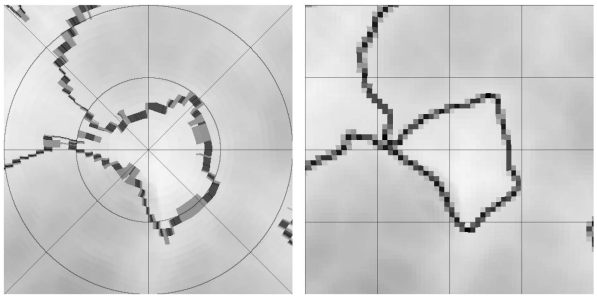
Given both spherical and polar data, we can produce uniform sampling planet-wide using terrain composition. To make this process concrete, let us return to our example planet. Fig. 15 shows the spherical projection of its surface color map. While the examples here show only the color map, the uniform handling of height and surface data enabled by our method extends this discussion to terrain geometry as well as any other surface mapped quantities, including the normal maps used to produce these figures.

Fig. 16a shows the south pole. The small contoured region there is stretched across the entire bottom of Fig. 15. We see the extremely nonuniform sampling resulting from the direct mapping of that image onto the sphere. Texels are compressed longitudinally, but not latitudinally. Optimal output texels should be square to properly represent the square samples of the source data.

So, we introduce the polar projection of the surface color map, as shown in Fig. 17. Fig. 16b shows this image mapped onto the sphere. Contrast the uniformly shaped pixels of Fig. 16b with the stretched pixels of Fig. 16a.

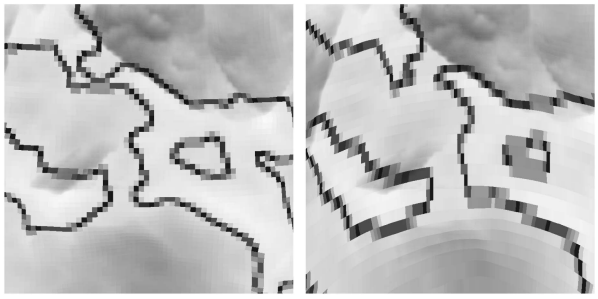
While polar data map cleanly at the pole, sampling suffers elsewhere. Contrast the uniformity of the spherical data near the equator, shown in Fig. 18a, with the nonuniform polar data at the same location in Fig. 18b.

To produce a uniform sampling across the entire planet, we must blend the spherical, north polar, and south polar



(a) (b)

Fig. 16. The south pole of the example planet, showing (a) spherical and (b) polar data mapped onto the sphere, contrasting the data sampling uniformity of each.



(a) (b)

Fig. 18. A region near the equator, showing (a) spherical and (b) polar data mapped onto the sphere, contrasting the data sampling uniformity of each.

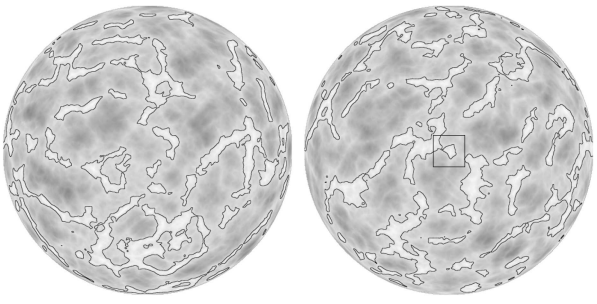
data sets. This blending follows immediately from the accumulation mechanism described in Section 2.3.2. The only open question is choosing the weights of that blend. There are alternative approaches here.

The figures in this paper use an extremely straightforward approach: cubic interpolation over distance. Fig. 19 shows each of the three weighted terms separately, with their composition shown on the right. In general, the weights need not add up to one. While (x, y, z) vectors or (r, g, b) color data are accumulated in the red, green, and blue channels of the buffer, the weights are accumulated in the alpha channel. The sum of the weights is then used to normalize the RGB value upon final rendering.

The straightforward blending over distance in this example depends upon the use of spherical and polar projection. Arbitrary projections including orthogonal and perspective projection may also be accommodated using a more powerful weighting function based upon screen-space derivatives.

Let (u, v) be the texture coordinate computed as a function of the inputs $(\theta, \phi, \bar{n}_x, \bar{n}_z)$ taken from the deferred texture buffer (Section 2.3.1). GLSL defines functions $dFdx$ and $dFdy$ giving the derivative of any GLSL variable with respect to the x- and y-axes of the target frame buffer, computed using forward or backward differencing. The sampling uniformity of a texel k may be computed as the ratio of the magnitudes of the gradients along each texture axis:

$$k = \sqrt{\frac{dFdx(u)^2 + dFdy(u)^2}{dFdx(v)^2 + dFdy(v)^2}}.$$



(a) (b)

Fig. 17. Polar projection of the diffuse color of the example planet. The marked region corresponds to Fig. 16. (a) North. (b) South.

The following function computes a weighting value α in $[0, 1]$, where texels mapping to squares in the output give 1 and texels mapping anisotropically to $n \times 1$ or $1 \times n$ in the output give 0:

$$\alpha = 1 - \left|\frac{\log\ k}{\log\ n}\right|.$$

The n parameter is a configurable quality coefficient. Setting $n = 2$ gives an extremely aggressive isotropy bias that allows only square pixels to make significant contribution to the accumulation. Depending on the degree of source data overlap, this may or may not be desirable. It will favor data viewed face-on and bias data mapped, for example, to the side of a mountain. For this reason, a less aggressive bias is usually preferable.

This weighting function is independent of the nature of the projection, and thus, may be used to blend arbitrarily projected data values on the basis of the *quality* of their projection on a per-texel basis. Applied during the surface accumulation phase (Section 2.3.2), it produces effects such as that shown in Fig. 19 automatically.

Applied during the geometry displacement phase (Section 2.2.5), it enables the adaptive composition of height values, giving high-quality geometry planet-wide. However, a bit of extra work is required. As is apparent in the vertex numbering, shown in Fig. 7, the layout of the geometry image buffer is logical rather than spatial. Neighboring vertices are not adjacent in this buffer. Because of this, the GPU's derivative functions (dFdx and dFdy) are not valid. Texture coordinate derivatives must explicitly be computed from the texture coordinates of logically adjacent vertices. This may be done during geometry generation.

3.2 Data Overlay

Terrain visualization frequently requires the overlay of unregistered height and surface maps of differing resolu-

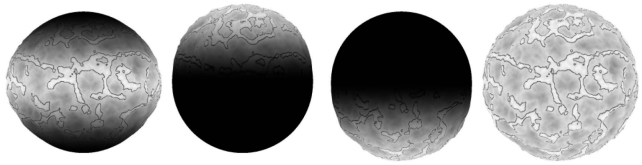


Fig. 19. The cubic-weighted contributions of spherical, north polar, and south polar projected height and color data to the final planet.

Fig. 20. A strip of local high-resolution data, as collected by a satellite in an inclined orbit.

tion and boundary. For example, one might need to view a high-res LIDAR height map of a fault in the context of the terrain where it lies.

Our collaborators include astronomers at Chicago's Adler Planetarium. As the Public Outreach and Education center for NASA's Lunar Reconnaissance Orbiter Camera mission (LROC), the Adler will receive 62TB of half-meter lunar imagery, beginning with the launch of LRO in May 2009. Our goal is to bring these data to the public via real-time 3D interactive experiences using the Adler's "Moon Wall" tiled display and 55-foot digital dome theater. To create a unique public interaction with these data, Adler astronomers wish to provide it as fresh as is possible. To form a coherent whole, gaps in coverage must be sealed with preexisting lunar data, such as the Clementine data set.

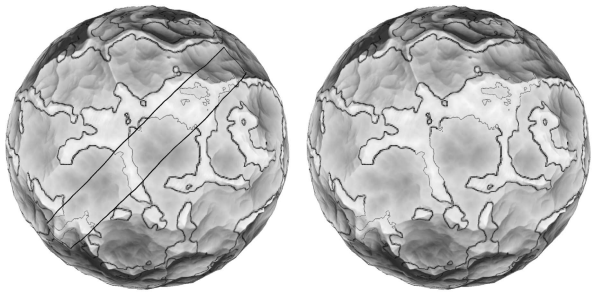
Data overlay is a multipass process. Height maps are written to the geometry accumulation buffer and surface maps are written to their accumulation buffers one by one. With each pass, the incoming (source) data may be composed with already-written (destination) data in a number of ways. If data passes are sorted from the lowest to the highest resolution, then the source may merely replace the destination, and the result represents the best available resolution at each point. Other applications may require the minimum or maximum value, or a blending of data sets based upon coverage, quality, or smoothing.

Detail data may be merged with base data by summing the source with the destination. This provides a solution to the specific problem of representing the *geoid*, the equipotential surface of the Earth taking into account its deviation from the sphere. As a specific example, NASA's MOLA data set optionally provides the Martian *areoid* in a height map separate from the Martian landscape. The sum of these gives the true radius of Mars at each point, correctly representing the planet's nonspherical shape.

As an example of localized data overlay, we have generated a projection of our sample planet similar to raw LROC output. The projection is an orthogonal strip scan around the planet, as if following the ground trace of a satellite in an inclined orbit. The color map appears in Fig. 20. It is outlined in place in Fig. 21a.

Texture coordinates are derived from the eye-space fragment position, with the inverse data projection applied. Height and surface maps are blended normally, but care is taken to clamp to the border of the overlaid input. Source fragments falling outside of the image are either discarded or masked away. Fig. 22a shows a close-up view of the border of the overlaid strip. Contrast the sampling of the contour line and note the discontinuity. If this abruptness is undesirable, then a blend function may be produced procedurally in the accumulation fragment shader, or encoded in the alpha channel of the image, as shown in Fig. 22b. This masking is fully generalized and overlaid data need not have rectangular boundary.

This technique may be optimized by confining rendering to the boundary of the overlaid data in the target buffer.



(a) (b)

Fig. 21. A strip of local high-resolution data composed with global low-resolution data. (a) With boundary indicated. (b) Composed normally.

When accumulating surface maps, one need draw only the screen-space shape filling the boundary. If this boundary is complex or expensive to compute, a screen-space rectangle or eye-space bounding volume will suffice. This results in fewer fragment operations than a full-screen pass. Similarly, when accumulating height maps, one need render only to those scan lines encoding the surface patches touched by the overlay.

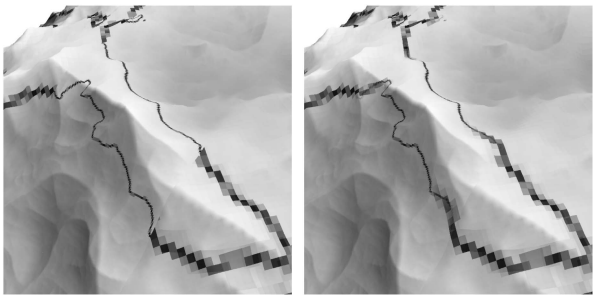
3.3 Level-of-Detail and Paging

Planetary-scale data sets continue to grow in extent and increase in resolution. Most data sets in use today far exceed the size of the available RAM or video RAM of the hardware used to display them.

To accommodate out-of-core data in real time, a caching mechanism must be used. A data set is sliced into manageable pages and down-sampled to a mipmap pyramid. We find a data tile size of 512×512 (510×510 plus a one pixel border) to give the best trade-off between granularity and throughput. The application uses the current view frustum to determine which of these pages are visible and at what resolution. Selected pages are uploaded to video memory for rendering under a least recently used policy.

Our implementation uses a variant on the technique developed by Lefohn et al. [15] to reference very high-resolution shadow maps. In this approach, texture coordinates do not map directly onto texels, instead they map onto a mipmap *index* texture, which contains references into a tile *cache* texture.

The cache texture behaves as a normal 2D image. It is an $n \times m$ *atlas* of all currently loaded data pages. Given a page



(a) (b)

Fig. 22. A close-up view of the border of the high-resolution strip of local data. Pixel size indicates sampling and resolution. (a) Unblended. (b) Blended.

size p, its size is $p \cdot n \times p \cdot m$. The maximum texture size of the GPU places an upper bound on n and m. Current hardware supports textures as large as 8,192 pixels square, and recent hardware supports 4,096 pixels. In our case of $p = 512$, we select $n = 16$ and $m = 8$, for a total of 128 cache lines. The parameters n and m serve to balance quality versus performance, as needed. The true tile size $p - 2$ is selected to be a power of two minus two in order to accommodate tile packing within an atlas of maximum size, while allowing a border for use in interpolation across tile boundaries.

The index texture behaves as a normal, unfiltered 2D mipmap. Rather than giving colors (r, g, b), this texture gives coordinates (r, c, l). These are the cache texture row r in $[0, n)$ and column c in $[0, m)$ of the page of data for the given texture coordinate, with the level-of-detail l of that page. The l parameter is used to recognize the presence of a lower resolution page to serve as proxy while the page of the desired resolution is being loaded. This virtual texture lookup process is performed by a GPU fragment shader, which may implement any mipmap access policy. This shader uses its knowledge of the cache size, page size, and border width to perform linear filtering explicitly, as normal OpenGL bordered texture sampling does not suffice in the context of an atlas. The figures and performance measures shown in this paper use a fragment shader implementation of trilinear mipmapping, which produces an optimal sampling of referenced data based upon texture coordinate derivatives.

The CPU maintains the state of both the index and cache textures. Each data set in use is represented by a quad-tree of page references. A rectangular data set will result in a full quad-tree, but fullness is not necessary. The National Elevation Database [29], which depicts U.S. territories all over the world, is an example of a sparse quad-tree.

Each page of this quad-tree has a rectangular shell bounding volume similar to that shown in Fig. 4. The solid angle subtended by this shell at the current viewpoint, combined with the resolution and solid angle of the display itself, allows the ratio of texels per pixel to be computed for each page. If this ratio meets a cutoff, then the corresponding page is asynchronously uploaded to the cache texture using a pixel buffer object. The page's cache location is uploaded to the index texture in both its correct position and any applicable proxy positions.

When a data set is applied, it is rendered as a single rectangle. During surface map rendering, a screen-sized rectangle is drawn. In the case of height map accumulation, the rendered rectangle covers the geometry accumulation buffer as shown in Fig. 8.

We will see this mechanism in action in our performance analysis, described in Section 4. There, it is used to provide real-time access to 115 GB of height, color, and normal map data.

4 Performance Measures

In an effort to quantify the performance of our algorithm and understand the variance in performance with different quality settings, we define a benchmark. A scripted 4,800-frame animation begins with a wide view of the Earth, moves in to a close-up view of Mount Rainier, and then moves back to the wide view along the same path. See Fig. 23. Each execution of this benchmark begins with an empty data cache and data are loaded as needed during the move in. On the move out, much of the needed data remain in the cache. This allows us to contrast the performance of the system under both I/O intensive and nonintensive circumstances. All frame time measurements are averaged over 10 frames, giving 480 data points per run.

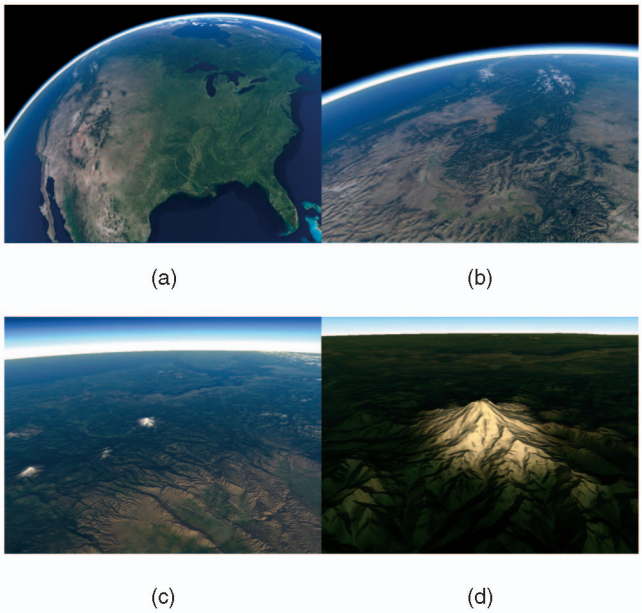


Fig. 23. Frames (a) 100, (b) 1,600, (c) 2,200, and (d) 2,400 of the 4,800-frame benchmark animation.

Our test configuration involves multiple gigapixel-scale data sets, paged and cached as described in Section 3.3, using a data overlay composition as described in Section 3.2. The base layer is the SRTM [11] data set covering the Earth at a resolution of 30 arcseconds, giving 3.5 gigapixels of 16-bit height data. The National Elevation Database (NED) [29] is overlaid atop this. NED covers all U.S. territory at a resolution of one arcsecond, giving 17 gigapixels of 16-bit height data. A 24-bit normal map is derived from each of these, enabling per-pixel illumination. Finally, the BMNG [22] data set provides 24-bit RGB color covering the Earth at 30 arcseconds, for another 3.5 gigapixels. In total, this is 115 GB of raw data. The data are preprocessed into 189,000 mipmapped, bordered tiles, each 512 pixels square. These tiles are compressed using PNG, consuming 38 GB of disk space. Rendering uses an atmospheric illumination model by O'Neil [23].

The character of the overlaid data is shown in Fig. 24. We see the low-resolution SRTM height and normal maps adjacent to the high-resolution NED height and normal maps, with the BMNG color map applied to both.

4.1 Baseline Performance

The primary test hardware is a dual AMD Opteron 250 at 2.4 GHz with 4 GB of RAM and an NVIDIA GeForce 8800 GTX. The baseline run of this configuration has a resolution of $1{,}024 \times 768$, a refinement depth $d = 4$, a triangle seed count $n_t = 256$. Data caches allow for 128 pages of 16-bit height data and 128 pages of 24-bit color and normal data. Results are shown in Fig. 25.

As expected, we see many page faults on the move in and only a few on the move out. There is a clear correlation between frame time and page fault count, indicating the

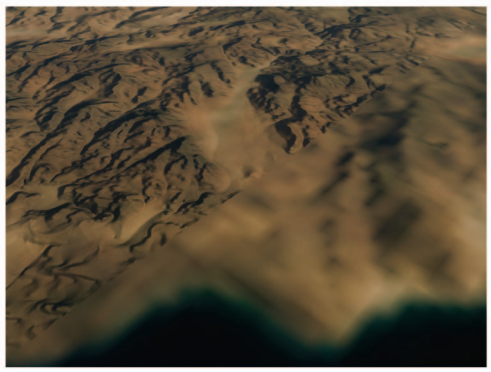


Fig. 24. The resolution discontinuity between the SRTM and NED height and normal maps, with the BMNG color map.

imperfect independence of the render thread from data loader threads due to communication overhead. On the move out, we see level performance around 6 ms per frame. This demonstrates the algorithm's consistent throughput and performance independent of proximity to the data set.

Note that all graphs presented here use the same vertical scaling, where the top of the graph represents a 30-Hz refresh rate and the middle of the graph represents a 60-Hz refresh rate. Fig. 25 shows performance comfortably better than 60 Hz throughout the run.

4.2 Variance with Resolution

Now we vary the resolution of the display without varying the complexity of the geometry. The results are shown in Fig. 26. Decreasing the display resolution to 320×240 (shown in blue) greatly reduces both fragment processing overhead and data demand. This reveals the combined overhead of the visibility and granularity phase (Section 2.1) and geometry generation phase (Section 2.2). We see performance level at 5 ms. This is the major fraction of the 6 ms performance at $1{,}024 \times 768$ (again in black), but it is fortunately a constant dependant only upon geometric complexity.

The primary impact of increasing the display resolution to $1{,}920 \times 1{,}080$ (Fig. 26 in red) is a higher demand for data. In this case, we see rough performance on the move out due to the reloading of low-resolution overview pages ejected during the high-resolution Mount Rainier close-up.

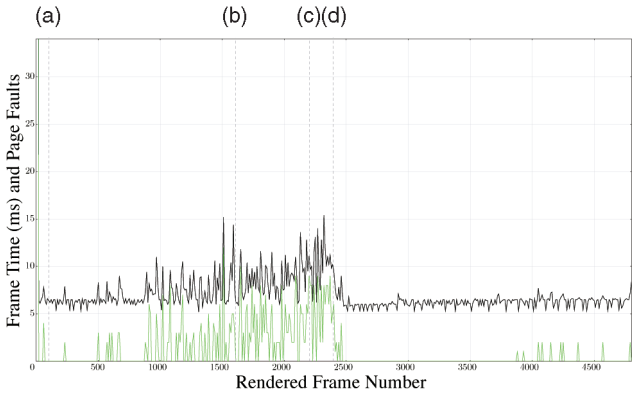


Fig. 25. Baseline performance at $1{,}024 \times 768$ ($d = 4, n_t = 256$) measured over time. Frame time (ms) is shown in black and page fault count in green. Marked frames (a), (b), (c), and (d) refer to Fig. 23.

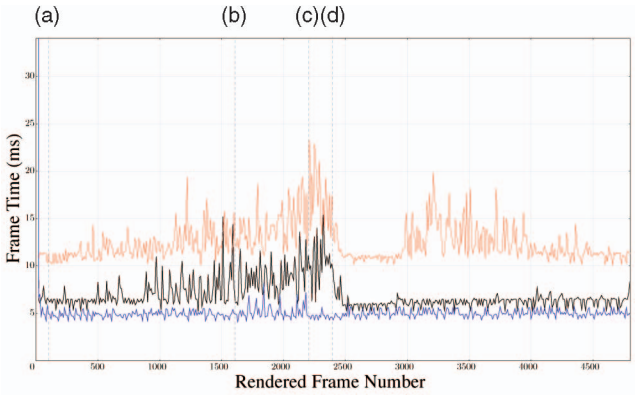


Fig. 26. Frame time (ms) measured over time at $1{,}024 \times 768$ in black, 320×240 in blue, and $1{,}920 \times 1{,}080$ in red.

We can still see a consistent minimum frame time of around 10 ms in this circumstance, due to fragment processing overhead. Given the assumption of 5 ms of geometry overhead inferred from the 320×240 results, we see the increasing fragment cost match the geometry cost at this resolution. As resolution increases from here, the balance tends toward fragment processing.

4.3 Variance with Geometry

Now we vary the parameters that determine geometric complexity while holding the display resolution constant at $1{,}024 \times 768$. First, we double the number of seed triangles from $n_t = 256$ to $n_t = 512$ while holding the refinement depth d constant. This has the effect of doubling the number of generated vertices from 39,168 to 78,336 (Section 2.1). We see this doubling borne out in Fig. 27 with the frame time graph translated upward.

Orthogonally to this, we increase the refinement depth from $d = 4$ to $d = 5$ while holding the seed triangle count n_t constant. This has the effect of increasing the number of generated vertices 3.6 times, from 39,168 to 143,616 (Section 2.2). Again, we see the impact of this clearly in Fig. 28. If we infer from Fig. 26 that 1 ms of the frame time may be attributed to the overhead of rendering $1{,}024 \times 768$ pixels, then the remaining 17 ms of $d = 5$ frame time is quite close to 3.6 times the 5 ms of $d = 4$ frame time.

It is worth noting that an increase in d quickly increases the vertex count, but does so without CPU cost. In contrast,

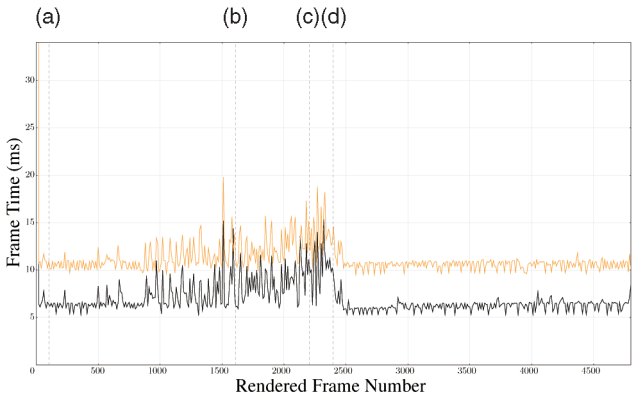


Fig. 27. Frame time (ms) measured over time at $1{,}024 \times 768$ with seed triangle count $n_t = 256$ in black and $n_t = 512$ in orange.

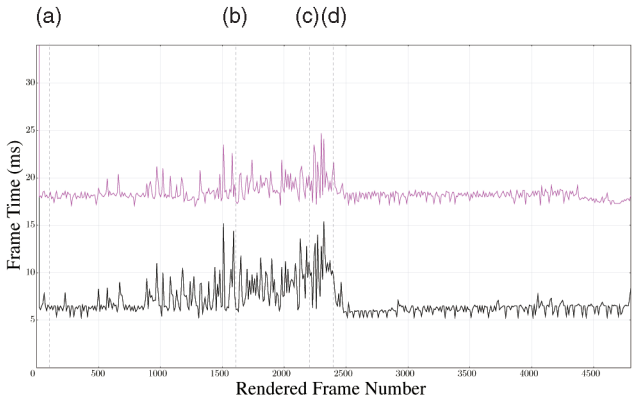


Fig. 28. Frame time (ms) measured over time at $1{,}024 \times 768$ with triangle subdivision depth $d = 4$ in black and $d = 5$ in pink.

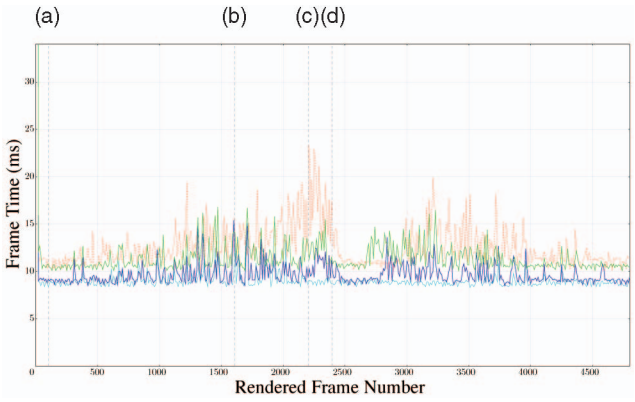


Fig. 29. Frame time (ms) measured over time at 1,920 $\times$ 1,080 with SRTM only in cyan, SRTM+NED height in blue, and SRTM+NED normal in green, with the baseline SRTM+NED height and normal in red (as in Fig. 26).

when doubling n_t, we double the CPU's geometry load in addition to doubling the vertex count. This incurs twice the visibility processing and twice the number of rendering batches. In our tests, we find the CPU load of the rendering thread to be negligible. However, one can imagine a circumstance, where careful manipulation of n_t and d may tune the CPU-GPU balance and improve throughput.

4.4 Variance with Data

The data caching mechanism ensures that the total size of a given data set does not impact general performance. However, data layering does incur overdraw, so performance does vary with the total number of data sets composed. To quantify this, we run our benchmark without the NED data set overlaid. This removes both the NED height map contribution from the terrain geometry and the NED normal map contribution from the normal buffer accumulation.

Fig. 26 showed that fragment processing is not a serious bottleneck at $1{,}024 \times 768$, so to see a distinction, we perform this test at $1{,}920 \times 1{,}080$. Fig. 29 shows this result in cyan. As we would expect, the SRTM-only frame time is reduced and performance is more consistent due to the lower data demand.

To see the impact of height map overdraw, we apply the NED height map atop the SRTM without the NED normal map. This is shown in Fig. 29 in blue. While these results are rougher due to data demand, the overall trend is not significantly slower than the SRTM-only performance. This indicates that the costs of height data resampling and geometry displacement are negligible.

Finally, to see the impact of surface map overdraw, we apply the NED normal map atop the SRTM without the NED height map. We see an overall increase in frame time of 2 ms in green. Surface map overdraw incurs the rendering of another rectangular screen-space bounding volume, and surface map accumulation is handled in fragment shading. Thus, the cost of applying further surface maps is linear in the number of fragments in the screen-space bounding volume of each.

In Fig. 29, the frame time shown in cyan is common to all four measurements. The difference between the blue and the cyan gives the cost of rendering the NED height map, and the difference between the green and the cyan gives the cost of rendering the NED normal map. The difference between the red and the cyan gives the cost of rendering both NED height and normal. We would expect this third difference to reflect the sum of the two others and it does not deviate far from the expectation.

4.5 Performance Qualities

These results reveal a balanced pipeline at our baseline configuration. Frame time increases with either an increase in geometry complexity or an increase in display resolution. This balance is in contrast with many of the approaches to terrain rendering presented in the literature. Triangle-pinching CPU-based algorithms in the style of ROAM [10] tend toward a geometric bottleneck, and well-batched GPU-based algorithms such as geomipmapping [8] lead to a pixel bottleneck. By offloading geometry processing to the GPU, we provide a mechanism to distribute the terrain processing cost more uniformly.

However, the situation becomes more complex when we look to the practice of multi-GPU rendering. PC motherboards with multiple PCI Express slots accepting more than one video board are common, as are single-slot video boards with multiple GPUs. These configurations require special attention, as independent GPUs have separate local VRAMs. Intermediate results, such as our generated geometry buffers and deferred shading buffers, may require synchronization.

We would be careless to overlook these issues, so Fig. 30 displays the results of early testing of our algorithm in a multi-GPU environment. Our baseline GeForce 8800 GTX is

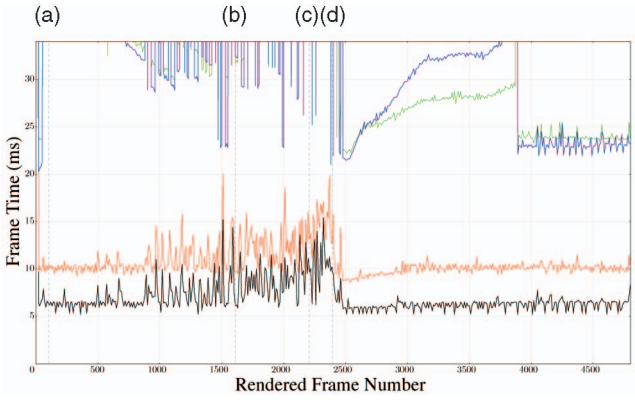


Fig. 30. Multi-GPU performance comparison: single GPU shown in black, multi-GPU in single mode in red, alternate-frame in green, split frame in blue.

shown in black, as above. Along side it, we see the performance of an NVIDIA GeForce 9800 GX2, which leaves a number of questions unanswered.

The performance of the 9800 with its multi-GPU capability disabled is shown in red. Frame time is longer by a consistent amount, indicating similar I/O response but decreased rendering throughput. Green shows the 9800 in alternate frame rendering (AFR) mode. In AFR mode, one GPU renders odd-numbered frames, while the other renders even-numbered frames. Despite the fact that our algorithm introduces no interframe data dependencies, the performance is radically altered. Blue shows the 9800 in split frame rendering (SFR) mode. In SFR mode, one GPU renders the top half of each frame, while the other renders the bottom half. Due to the amount of intraframe data dependency in our algorithm, we would expect this mode to perform the worst, and it does.

However, both AFR and SRF modes suffer from pathologically bad I/O response time. The results of the first half of the benchmark are effectively unusable. Frame time eventually levels off at 23 ms per frame late in the non-I/O-intensive second half of the benchmark. This performance degradation could be explained by the synchronization penalty, but we would expect this to begin around frame 2,500. Its slow crawl from 23 ms up to and beyond 30 ms during this period remains unexplained.

In the context of standard forward rendering, the current hardware industry trend toward multi-GPU configurations is clearly beneficial. But given the tendency of modern GPU algorithms to utilize render-to-texture and other data-dependent techniques, the results shown here are troubling. With a conflict between the functionality that the software uses and the functionality that the hardware provides, an important area for future work is revealed. Until such issues are resolved, we must utilize the parallel rendering capability of our implementation to treat multiple GPUs as wholly separate renderers, each with an independent frame buffer.

5 Conclusions and Future Work

In this paper, work we have demonstrated a mechanism for the real-time manipulation and display of terrain height and surface data. Beyond simply rendering terrain, this mechanism affords opportunities to combine data in powerful ways, bringing together disparate planetary-scale data sets smoothly and efficiently, and adapting to produce a uniform composite visualization of them.

Throughout our discussion, we have detailed a number of areas where future work is required. In particular, we must analyze the error and the effects of nonuniform sampling of height data. In addition, while the generality of terrain composition provides a path to an elegant implementation of geomorphing, we have yet to implement it. Finally, the adaptation of our highly data-dependent technique to multi-GPU environments will be increasingly significant as this hardware trend continues.

Despite these remaining issues, the established ability to perform arbitrary manipulation and blending of planetary data has unlimited application. A number of further composition operations have been proposed and remain to be explored.

One potential composition involves color-space transformation. The Mars Reconnaissance Orbiter (MRO) HiRISE camera provides extremely high-resolution imagery of Mars, but not in full color. In contrast, the Viking Orbiter image data set provides full color, but at low resolution. A real-time composition may combine the low-res chroma of Viking Orbiter with the high-res luminance of MRO, giving high-resolution photo-realistic imagery of Mars.

Another potential composition addresses issues of data layering. Data exist representing distinct surface layers and subsurface geological structure, and users seek to visualize these layers in the context of one another. Examples include Antarctica, where both the surface of the ice and the surface of the land are of interest. Arbitrary blending and masking of data enables the overlay of these, affording a tool to peel away layers of data along arbitrary user-selected boundaries.

Finally, the display of time-varying data stored as a sequence of images is trivially expressed in terms of terrain composition. Consecutive key frames may be smoothly interpolated without the need to generate and store intermediate frames. The resulting smoothed animation is applicable to both height geometry and surface map data.

Our partnerships with the AGIC and the Adler Planetarium will continue to drive the investigation into these types of operations. Their access to new large-scale data sets will raise new requirements, leading to as-yet-unforeseen formulations of terrain composition.

Acknowledgments

The Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago specializes in the design and development of high-resolution visualization and virtual-reality display systems, collaboration software for use on multigigabit networks, and advanced networking infrastructure. This paper is based upon work supported by the US National Science Foundation (NSF) awards CNS-0420477, CNS-0703916, OCI-0441094, OCI-0225642, OCE-0602117, and DRL-0426328, and the NASA ASTEP Program award NNX07AM88G. EVL also receives funding from the State of Illinois, Adler Planetarium, Science Museum of Minnesota, Office of Naval Research on behalf of the Technology Research, Education, and Commercialization Center (TRECC), Sharp Laboratories of America, and Pacific Interface on behalf of NTT Network Innovation Laboratories in Japan. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies and companies.

References

[1] *Globalmapper,* http://www.globalmapper.com/, 2009.
[2] T. Boubekeur and C. Schlick, "Generic Mesh Refinement on GPU," *Proc. ACM SIGGRAPH/EUROGRAPHICS,* pp. 99-104, 2005.
[3] T. Boubekeur and C. Schlick, "A Flexible Kernel for Adaptive Mesh Refinement on GPU," *Computer Graphics Forum,* vol. 27, no. 1, pp. 102-113, 2008.
[4] E. Bruneton and F. Neyret, "Real-Time Rendering and Editing of Vector-Based Terrains," *Computer Graphics Forum,* vol. 27, no. 2, pp. 311-320, 2008.
[5] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM)," *Proc. IEEE Conf. Visualization,* pp. 147-155, 2003.
[6] M. Clasen and H.-C. Hege, "Terrain Rendering Using Spherical Clipmaps," *Proc. Eurographics/IEEE-VGTC Symp. Visualization,* 2006.

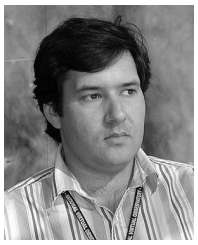
[7] C. Dachsbacher and M. Stamminger, "Rendering Procedural Terrain by Geometry Image Warping," *Rendering Techniques,* pp. 103-110, 2004.
[8] W.H. de Boer, "Fast Terrain Rendering Using Geometrical Mipmapping," http://www.flipcode.com/, Oct. 2000.
[9] J. Döllner, K. Baumman, and K. Hinrichs, "Texturing Techniques for Terrain Visualization," *Proc. IEEE Conf. Visualization,* pp. 227-234, 2000.
[10] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineev-Weinstein, "ROAMing Terrain: Real-Time Optimally Adapting Meshes," Technical Report UCRL-JC-127870, Lawrence Livermore Nat'l Laboratory, Oct. 1997.
[11] T. Farr and M. Kobrick, "The Shuttle Radar Topography Mission," *Rev. Geophysics,* vol. 45, no. 2, 2005.
[12] R. Ferguson, R. Economy, W. Kelly, and P. Ramos, "Continuous Terrain Level of Detail for Visual Simulation," *Proc. IMAGE V Conf.,* pp. 144-151, 1990.
[13] X. Gu, S. Gortler, and H. Hoppe, "Geometry Images," *ACM Trans. Graphics,* vol. 21, no. 3, pp. 355-361, 2002.
[14] L.M. Hwa, M.A. Duchaineau, and K.I. Joy, "Adaptive 4-8 Texture Hierarchies," *Proc. IEEE Conf. Visualization,* pp. 219-226, 2004.
[15] A.E. Lefohn, S. Sengupta, and J.D. Owens, "Resolution-Matched Shadow Maps," *ACM Trans. Graphics,* vol. 26, no. 4, p. 20, 2007.
[16] J. Levenberg, "Fast View-Dependent Level-of-Detail Rendering Using Cached Geometry," *Proc. IEEE Conf. Visualization,* pp. 259-265, 2002.
[17] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, A. den Bosch, and N. Faust, "An Integrated Global GIS and Visual Simulation System," Graphics, Visualization, and Usability Center, Georgia Inst. of Technology, 1997.
[18] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner, "Real-Time, Continuous Level of Detail Rendering of Height Fields," *Proc. ACM SIGGRAPH,* pp. 109-118, 1996.
[19] P. Lindstrom and V. Pascucci, "Visualization of Large Terrains Made Easy," *Proc. IEEE Conf. Visualization,* pp. 363-574, 2001.
[20] Y. Livny, N. Sokolovsky, T. Grinshpoun, and J. El-Sana, "A GPU Persistent Grid Mapping for Terrain Rendering," *The Visual Computer,* vol. 24, no. 2, pp. 139-153, 2008.
[21] F. Losasso and H. Hoppe, "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids," *Proc. ACM SIGGRAPH,* pp. 769-776, 2004.
[22] NASA, *Blue Marble Next Generation,* http://worldwind.arc.nasa.gov/bluemarble.html, 2005.
[23] S. O'Neil, "Accurate Atmospheric Scattering," *GPU Gems,* vol. 2, chapter 16, Addison-Wesley, 2005.
[24] R. Pajarola and E. Gobbetti, "Survey of Semi-Regular Multiresolution Models for Interactive Terrain Rendering," *The Visual Computer,* vol. 23, no. 8, pp. 583-605, 2007.
[25] K. Perlin, "Standard for Perlin Noise," US Patent #6,867,776, Mar. 2005.
[26] M. Reddy, Y. Leclerc, L. Iverson, N. Bletter, S. Int, and M. Park, "TerraVision II: Visualizing Massive Terrain Databases in VRML," *IEEE Computer Graphics and Applications,* vol. 19, no. 2, pp. 30-38, Mar./Apr. 1999.
[27] T. Saito and T. Takahashi, "Comprehensible Rendering of 3D Shapes," *ACM SIGGRAPH Computer Graphics,* vol. 24, no. 4, pp. 197-206, 1990.
[28] J. Schneider and R. Westermann, "GPU-Friendly High-Quality Terrain Rendering," *J. WSCG,* vol. 14, nos. 1-3, pp. 49-56, 2006.
[29] *National Elevation Database,* EROS Data Center, US Geological Survey (USGS), http://ned.usgs.gov/, 1999.

Robert Kooima received the PhD degree in 2008 from the University of Illinois at Chicago, where he was a graduate research assistant at the Electronic Visualization Laboratory. His research interests include real-time 3D graphics and large-scale display technology. He is a member of the IEEE.

Jason Leigh is an associate professor of computer science and the director of the Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago. He is a cofounder of VRCO, the GeoWall Consortium, and the Global Lambda Visualization Facility. He currently leads the visualization and collaboration research on the National Science Foundation's OptIPuter project and has led EVL's TeleImmersion research agenda since 1995. His main area of interest is in developing collaboration technologies and techniques for supporting a wide range of applications ranging from the remote exploration of large-scale data, education, and entertainment.

Andrew Johnson is an associate professor in the Department of Computer Science and a member of the Electronic Visualization Laboratory at the University of Illinois at Chicago. His research focuses on the development and effective use of advanced visualization displays, including virtual reality displays, auto-stereo displays, and high-resolution walls and tables, for scientific discovery and in formal and informal education. He is a member of the IEEE and the IEEE Computer Society.

Doug Roberts manages the NUIT Vislab at Northwestern University, which supports researchers and educators use stereoscopic and ultra-high resolution displays to view complex data sets. This visualization experience is put to use at Adler, which is a group of astronomers, educators, artists, and technologists that brings immersive visualization experiences to the public in a newly constructed space on the exhibit floor. He is the director of a new Space Visualization Laboratory at Adler.



Mark SubbaRao serves as the director of Visualization for the Adler's Space Visualization Laboratory, where he supervises the development of scientific visualizations and interaction techniques for the cutting edge stereoscopic and ultra-high resolution displays in the laboratory. His area of research is cosmology, particularly the large-scale structure of galaxies, their clustering properties, and evolution. He is a member of the Sloan Digital Sky Survey, where he is a developer of spectroscopic software that automatically measures redshifts, hence, 3D positions, for nearly 1,000,000 galaxies and quasars.

Thomas A. DeFanti is a research scientist at the California Institute for Telecommunications and Information Technology (Calit2), University of California, San Diego. He was the director of the Electronic Visualization Laboratory, University of Illinois at Chicago. He shares the recognition along with EVL director Daniel J. Sandin for conceiving the CAVE virtual reality theater in 1991. Striving for more than a decade to connect high-resolution visualization and virtual reality devices over long distances, he has collaborated with Maxine Brown of UIC to lead state, national, and international teams to build the most advanced production-quality networks available to academics.