Planetary-scale Terrain Composition

Robert Kooima

November 3, 2008

Contents

Acknowledgements

Thank you to the Adler Planetarium and Astronomy Museum for partial support of this work and use of the Definiti Space Theater and facilities during the defense. Thank you to Doug Roberts and Mark SubbaRao of the Adler for serving on my committee.

Thanks to the California Institute for Telecommunications and Information Technology for partial support of this work and for significant exposure through the use of the Varrier and StarCAVE virtual reality environments.

Thanks to Paul Morin of the University of Minnesota's Antarctic Geospacial Information Center for nearly a terabyte of source data used in this work.

Finally, thank you to everyone at the Electronic Visualization Laboratory for welcoming me into their community. Thanks to Jason Leigh and Andy Johnson for their leadership and advising. Thanks especially to Tom DeFanti and Dan Sandin for their continued mentoring.

Summary

Many inter-related 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. I 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 universally applied to both. Examples include adaptation to non-uniform 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 sub-meter scales.

Abbreviations

AFR	Alternate Frame Rendering
BMNG	Blue Marble Next Generation
CUDA	Compute Unified Device Architecture
GLSL	OpenGL Shading Language
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
LIMA	Landsat Image Mosaic of Antarctica
LOD	Level of Detail
LROC	Lunar Reconnaissance Orbiter Camera
MOLA	Mars Orbiter Laser Altimeter
NED	National Elevation Dataset
ROAM	Real-time Optimally Adaptive Mesh
SFR	Split Frame Rendering
SRTM	Shuttle RADAR Topography Mission
USGS	United States Geological Survey
VRAM	Video RAM (Random Access Memory)

Chapter 1

Introduction

1.1 Motivation

A vast quantity of data exists describing the Earth and other planets, their terrain and features, their surface reflectance in many wavelengths, and a variety of other quantities sampled across their land-forms. Thanks to the many sensors and instruments currently deployed and in development, this quantity of data is expanding at an ever increasing rate. These data find use in a wide variety of disciplines including geology, geography, climatology, astronomy, planetary science, and more.

The format, type, projection, resolution, and coverage of these data sets are appropriate to their subject, and thus are as varied as the subjects. However, all of the data relating to given land-form *are* related, and there is a clear motivation to bring these data together in a common visualization.

Unfortunately, terrain visualization literature does not emphasize the *composition* of disparate data sets. The brute-force approach is assumed: if you want to juxtapose multiple elements, you merely render each in turn. The extent to which the composition of elements is addressed usually goes no further than simply texturing an image data set onto the geometry of a height data set.

We are motivated then to consider mechanisms for enabling more powerful data composition operations. Here are some examples.

- Neither spherical nor polar projections suffice to usefully represent global data (Section 2.8.2). To render the entire globe, at least three distinct data sets should be used. We need a means to seamlessly combine dissimilar projections, adapting to the areas of optimal sampling of each.
- Data coverage may not be of uniform resolution. For example, a high resolution map of a fault line may be displayed in the context of the terrain where it lies. It is of no benefit to re-sample one to the resolution of the other, so when surface geometry is generated we may make no assumptions of granularity and must smoothly accept transitions in resolution.
- As a visualization zooms, the granularity of the geometry must adapt. If significant care is not taken, these adaptations occur suddenly and the output pops from one level of detail to the next. However if geometry is operated upon as imagery, then discontinuous level-of-detail transitions may be smoothed using simple image blending.
- Realistic rendering benefits from the use of per-pixel illumination processing using bump texture mapping. However, such textures are derived from height maps, which are not necessarily registered with the image map giving the diffuse surface color. We need a means of adaptively registering the inputs to the lighting model with one another.
- Surface reflectance sensors measure in limited wavelengths, giving science data, but not necessarily giving visual realism. If for example the

high resolution luminance of Mars Reconnaissance Orbiter (MRO) HiRISE were combined with the low-resolution chroma of Viking Orbiter, a high-resolution photo-real image of Mars would result. To support this, we need the ability to transform input color-space, register data sets, and transform output to RGB on the fly.

- Many data sources are time-variant. For example, Blue Marble Next Generation (BMNG) provides a separate 3 giga-pixel image of the Earth for each of the 12 months of the year 2004. The BMNG images prior to and following any given date may be blended, giving a smoothly-varying approximation of the appearance of the Earth at any time in 2004. Thus we can produce an animation spanning the year without popping from month to month.

- Shaded relief is a non-photorealistic topographical representation highly valued by geologists. Users insist upon precise control over lighting parameters and tinting, while still retaining the ability to merge relief shading with other geologic data. We desire a means to extract relief from height maps and illuminate it with interactive control.

- Many data are layered. For example, Antarctica has both an ice layer and a bedrock layer. We need the means to peel these structures apart and see their layers in relation to one-another.

Enabling such operations flexibly and interactively elevates terrain rendering from a static visualization tool to a dynamic query and exploration tool. Providing the means to integrate arbitrarily large data stores in this fashion in real-time leads to new ways of working with the streams of data flowing in from throughout the solar system. Recent advances in graphics processing hardware make these possible.

1.2 Moving forward

I propose a GPU-centric real-time approach to generating and rendering planetary bodies composed of arbitrary quantities and types of height-map data, textured and illuminated using arbitrary quantities and types of surface-map data. At the core of this approach lies the assertion that modern graphics hardware need not draw a distinction between colors and vectors. Massively parallel vector stream processors allow color computation to be performed with 32-bit floating point precision, merging the processing of color with that of geometry. With the distinction between color and vector blurred, a variety of highly efficient image processing operations become applicable to both terrain geometry and terrain surface maps.

We begin the discussion of this approach with Chapter 2, an overview of the history of terrain rendering and the 3D hardware that made it possible, including a discussion of current hardware capabilities. These capabilities set the stage for the main focus of this work, the proposed terrain rendering approach documented by Chapter 3. Chapter 4 then examines in detail a few terrain composition techinques enabled by this algorithm. Finally, we validate the approach with an examination of an implementation of the algorithm that demonstrates all of these concepts. We see its use in a parallel real-time composition of 115GB of Earth data, with a quantitative analysis of the algorithm's behavior under this load in Chapter 5.

Chapter 2

Terrain rendering and hardware evolution

The field of terrain rendering algorithm research is vibrant, and has been for many decades. Traditional motivations dating back to the birth of computer graphics focus on flight simulation as the driving application for real-time algorithms. Since then real-time visualization has been embraced by broad communities of scientists including geographers, geologists, cartographers, and planetary scientists. Today, rendered terrain has spread to the entertainment industry, becoming ubiquitous in film and 3D gaming, forming the substrate of virtual worlds of all types.

The simplicity of basic terrain rendering and the satisfying output of even a modest effort lends the field a wide appeal. Invariably, all 3D graphics professionals, researchers, and enthusiasts approach the problem of terrain rendering at some point in their careers, often at a very early stage. This has led to a proliferation of implementations and approaches.

In this chapter we will review several notable terrain rendering algorithms. Each approach will be described and we will examine their strengths and weaknesses. Most significantly, we will see how each is motivated primarily by the capabilities of the hardware of the day. This will lead us to examine the capabilities of current and future hardware, and finally to extrapolate the influence this may have on terrain rendering.

Again, the field of terrain rendering is vast, and this discussion is by no means a complete review of it. It is instead a presentation of the dominant *forms* of algorithms used throughout the long history of the field, with specific regard to the interplay of influence between hardware and software which guided it.

We will begin by examining the regular grid, the simplest possible approach to mapping height onto geometry. From there, we will see how triangulated irregular network approaches preprocess these grids, optimizing them for better performance on early graphics hardware. Then, we will see this static optimization replaced by ROAM, a dynamic optimizing approach that adapts to the viewer. The arrival of powerful GPUs enables a shift from CPU-centric algorithms toward GPU-centric algorithms, such as geomipmapping. Finally, the fully programmable GPU enables the geometry clipmap approach to terrain geometry rendering. Having arrived at the state of the art, we touch upon additional background material surrounding realism, and the application of terrain rendering techniques to spherical planets.

2.1 Regular grids

2.1.1 Motivation

In Figure 2.1 we see the basic 3D rendering pipeline, loosely as defined by the original OpenGL specification in 1992 [36].

The application submits geometry in the form of vertices, normals, texture coordinates, and materi-

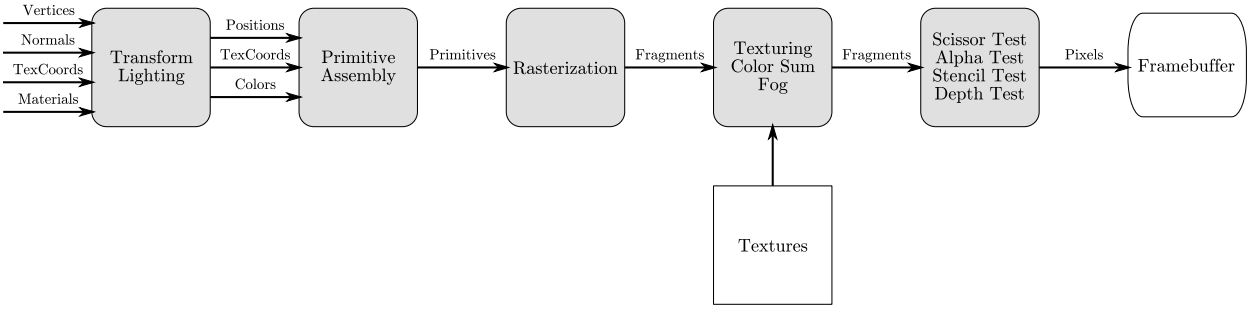


Figure 2.1: The OpenGL fixed-function pipeline. (Gray areas are fixed; white areas are user-controlled.)

als. View and projection transformations are applied to these, and lighting is computed. Polygons are assembled into triangular primitives with a position, texture coordinate, and color at each vertex. These attributes are interpolated across the face of each triangle during rasterization, and each resulting fragment is textured and tested for inclusion in the frame buffer.

2.1.2 Grid rendering

Given a two-dimensional block of elevation values, submitting a height map to the 3D pipeline is straightforward. The (x, y) position of the center of each pixel in the height map, with the z position taken from the pixel's value, gives a coordinate (x, y, z) in 3D space, which is biased and offset as needed to achieve the desired scale.

Triangles are wound from the grid as in Figure 2.2, with an $n \times m$ height map giving an $n-1 \times m-1$ array of right triangles. The result is $2(n-1)(m-1)$ triangles connecting $n \cdot m$ vertices. Normal vectors for each triangle are computed from vertex positions, and the normal vector for each vertex is the average of those of all adjacent triangles.

Given that the transform-lighting ("T&L") stage of the pipeline must process each of these $n \cdot m$ vertices in turn, a common optimization submits geometry to the 3D pipeline in the form of *triangle strips*, linear sequences of adjacent triangles, each sharing one edge with the previous. As shown in Figure 2.3, this reduces the total vertex processing cost to $2n(m-1)$.

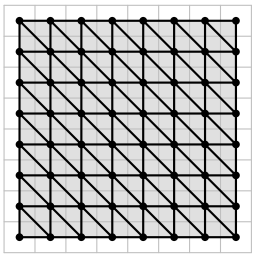


Figure 2.2: An 8×8 height map gives a 7×7 triangle grid.

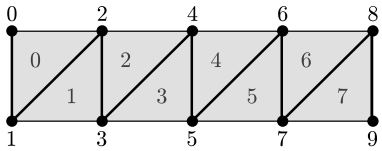
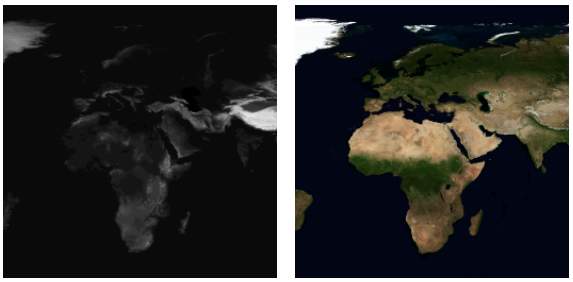


Figure 2.3: A *triangle strip*, efficiently defining n triangles using $n+2$ vertices

The application of a color map texture to this geometry is equally straightforward. It is commonly assumed that the color map is registered with the height

map. That is, while the two maps are not assumed to have the same image size, they are assumed to have the same surface coverage. See Figure 2.4. Under these circumstances, the texture coordinate (s, t) of each vertex is merely the pixel center (x, y) divided by the height map size to the range $[1, 1]$.



(a) Height map (b) Color map

Figure 2.4: Registered color and height maps have identical surface coverage

This registration assumption is near universal in terrain rendering literature. In most cases, color map application is taken for granted or relegated to an afterthought, barely worthy of mention. This is not unreasonable given the relative efficiencies of geometry processing versus texture processing.

2.2 Triangulated irregular networks

2.2.1 Motivation

Early research in terrain rendering focused on the task of geometry minimization. The goal was to determine the minimum number of triangles necessary to represent a given land-form to within some measure of precision. This goal was motivated by the computational cost of each individual geometry element. Circa 1995 a high-end graphics workstation could display a scene consisting 10,000 triangles at 30Hz, so the 2M triangles of a 1K×1K regular grid were out of the question.

2.2.2 Algorithm

The most widely-cited grid simplification approach is that of Garland and Heckbert [12]. A survey of similar algorithms compiled by the same authors accompanies that document [13]. The algorithm is greedy and proceeds iteratively, beginning with a flat plane approximating the surface, and adding vertices and triangles as needed until a minimum error bound is met.

At each step, every point of the input grid is compared against the current surface approximation. The point that deviates the farthest from the surface is added to the output, and a *Delaunay triangulation* of the current set of output points is computed, giving the refined surface approximation. This repeats until no point deviates farther than the desired error bound.

As background, a Delaunay triangulation of points maximizes the minimum angle of each triangle in an attempt to avoid thin sliver triangles.

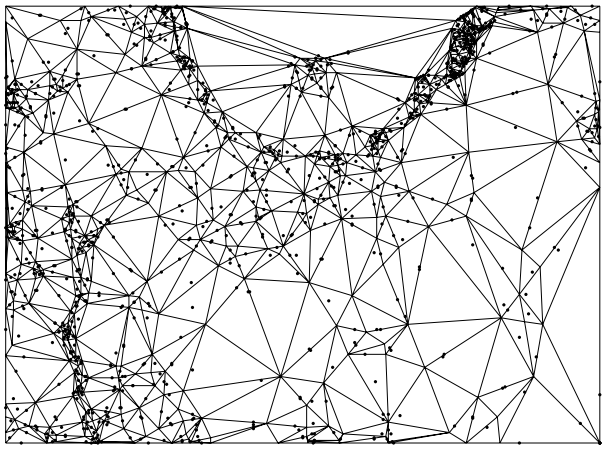


Figure 2.5: A triangulated irregular network near Crater Lake, with many triangles in areas of high detail

Figure 2.5 shows an example output Delaunay triangulation consisting of 555 vertices from an input grid 336×459 in extent. The dots in this figure denote candidate points for potential insertion, an optimization of the basic algorithm using a priority queue

to reduce the cost of searching the input data for the next refinement.

2.2.3 Impact

Triangulated irregular networks are very effective at minimizing the geometry cost of rendering. The refinement algorithm succeeds in seeking out the areas of high detail in the input land-form, and focusing geometry there. However, this happens at significant preprocessing expense. Triangulated irregular networks are appropriate for static data sets rendered at a predefined resolution, though they do not allow for dynamic data or run-time refinement.

2.3 ROAM

2.3.1 Motivation

When attempting to minimize geometry, there is a computational trade-off to be made. From any given view point, an optimal triangulation of a surface places many small triangles near the viewer, and a few large triangles farther away. This is referred to as *level-of-detail*, and is a basic tenet of real-time optimization. LOD algorithms attempt to ensure that each on-screen triangle makes a similar contribution to the complexity of the scene, regardless of that triangle's position in virtual space. Unfortunately, updating the working geometry for each frame adds an additional computational cost on top of the basic cost of rendering.

2.3.2 Algorithm

The canonical example of this type of system is Mark Duchaineau's Real-time Optimally-Adapting Mesh (ROAM) [5]. The ROAM algorithm successfully manages level-of-detail and visibility determination while exploiting frame-coherence, the frame-to-frame similarity in the solution resulting from the assumed smooth motion of the viewer.

In Figure 2.6 we see an explanatory representation of the output of the ROAM algorithm. The viewer is at the left, looking to the right, with light-colored triangles falling within the field of view. The small triangles near the viewer indicate the satisfaction of the level-of-detail criterion. In addition, the triangulation reflects the distribution of detail in the true surface by simplifying planar neighborhoods using larger triangles, as would a triangulated irregular network.

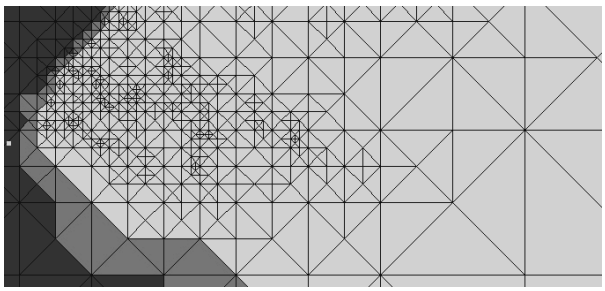


Figure 2.6: ROAM algorithm adaptive triangulation. The viewer is at the left, looking right.

To accomplish this, ROAM begins by tessellating the height field using right triangles and preprocessing these to determine a hierarchical *bintree*, Figure 2.7. Each node of this hierarchy stores an error metric quantifying the deviation of the triangulation from the true surface over the same area. Fine-grained nodes deep in the hierarchy approach the resolution of the height field and have small error values. Coarse nodes near the root represent the surface at lower levels of detail and have large errors.

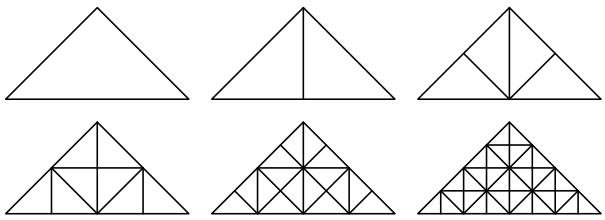


Figure 2.7: A *bintree*, a recursive subdivision of right triangles

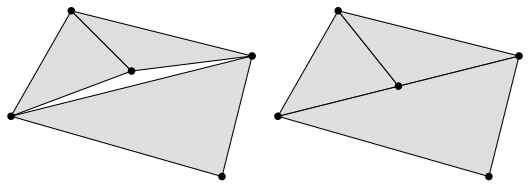
At run time, ROAM maintains a set of the bintree nodes giving a current triangulation. A pair of

priority queues track the largest and smallest error values of the current bintree node set, biased by the on-screen size of each node. Each time the view point changes the highest priority nodes are split (replaced by their bintree children) in order to maintain a minimum error bound. The lowest priority nodes are merged (replaced by their bintree parents) in order to maintain a consistent triangle count.

The hierarchical nature of the geometry representation leads to an efficient visibility mechanism. A bintree node is tested against the planes defining the field of view. If the node is entirely inside or outside of the view, then all of its children may be assumed to be inside or outside respectively. If the node is partially visible, the test proceeds recursively to the children. This approach is $O(\log n)$ in the extent of the height field.

2.3.3 Caveat: T-intersections

When tessellating a surface, care must be taken to ensure that adjacent triangles meet only at their vertices. In Figure 2.8a we see an erroneous configuration highly likely to occur during adaptive tessellation. The fine-grained geometry does not meet the coarse-grained, leaving a gap in the triangulation. Adjusting the vertex as in Figure 2.8b would seem to fill the gap, but the problem remains unresolved. When rasterized, interpolation along the long edge of the large triangle is not guaranteed to touch the same set of pixels as the interpolation along the two short edges. The result is a sparkle of random pixels along the edge. This condition is known as a *T-intersection*.



(a) Unaligned (b) Aligned

Figure 2.8: T-intersections. Both aligned and unaligned are to be avoided

From Figure 2.6, it is clear that ROAM solves this problem elegantly. It merely mandates that spatially-adjacent bintree nodes be within one level of each-other in the hierarchy. Given this restriction, the bintree ensures that no vertex can fall along the edge of an adjacent triangle.

2.3.4 Impact

The ROAM algorithm does an excellent job of maintaining a consistent triangulation in real-time. However its preprocessing requirement mandates an in-core data source, limiting its scalability. Performance testing presented in [5] was performed using a Silicon Graphics Onyx, the most powerful 3D hardware of the day, and the algorithm achieved 6,000 triangles per frame at 30Hz, given a 1K height field. While impressive at the time, these numbers would soon be eclipsed.

2.4 Geomipmapping

2.4.1 Motivation

By the late 1990s, the consumer-grade graphics hardware industry had begun its rapid advance. The increase in competition within the industry led to significant additional capability and capacity, and the accompanying drop in hardware prices brought widespread adoption. Real-time 3D algorithms adapted to take advantage.

A significant departure from previous hardware generations was the addition of hardware-based geometry processing, commonly known as "hardware T&L." This allowed processing such as transformation matrix application and lighting computation to be off-loaded from the CPU, to be performed instead by the graphics hardware itself.

At the same time, with more video RAM available, efficiency could be improved by storing geometry data in VRAM rather than main RAM. This would eliminate most per-frame geometry specification and reduce CPU-GPU communication, an increasingly problematic bottleneck.

Together, these influences caused CPU-centric vertex and triangle generation algorithms to pass out

of style. A brute-force off-loading of geometry to VRAM could easily outperform the most careful geometric algorithm simply because a balance had shifted. It became cheaper to render 100,000 triangles directly from VRAM than to select 10,000 optimal triangles to transfer from main RAM.

Focus turned to the concept of *batching*, and a classic granularity trade-off emerged. We define a *batch* as a static set of geometry, stored in VRAM, rendered as an atomic unit. A large batch may make optimal use of the GPU, but its geometry may extend beyond the view frustum, resulting in the processing of unseen data. A large number of small batches may allow for effective visibility testing, but will reduce total throughput.

An influential analysis by Matthias Wloka of the trade-offs inherent in batching [45] resulted in a best-practices target of only 300 batches per frame when rendering at 30Hz using a 2GHz CPU. More batches would lead to a CPU bottleneck, and fewer would leave idle GPU. Of course this number is expected to increase as CPU power (or more precisely, bus bandwidth) increases, but it underscores the fundamental transition in the philosophy of the design of real-time algorithms for the hardware of the day: previously, 6,000 rendered units was maximal, but after a hardware *advance* 300 rendered units was maximal, and the complexity of each unit is increased.

2.4.2 Algorithm

Several distinct terrain rendering approaches have been proposed that address this transition in similar fashion. These approaches will be collectively referred to here as *Geomipmapping* algorithms [4]. This name draws an appropriate parallel to the traditional *mipmapping* [44] approach to texture level-of-detail.

A mipmapped texture stores a series of source images subsampled to successive powers of two (Figure 2.9). When the texture is referenced during rasterization, screen-space texture coordinate derivatives determine which of these source images most closely matches the resolution of the display, and the corresponding pixel from the optimal image is used, thus achieving a level-of-detail optimization.

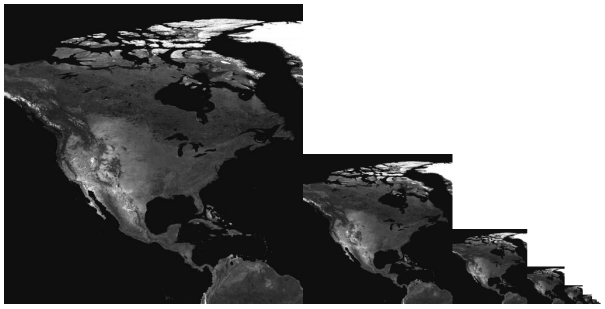


Figure 2.9: A *mipmap*, a pyramid of images subsampled at powers of two

Similarly, a geomipmapped geometry representation stores a *quad-tree* hierarchy of surface batches (Figure 2.10). Each node of a geomipmap hierarchy batches a similar amount of geometric detail, but a given child node covers a quarter of the surface area of its parent node. Taken together, each individual layer of the hierarchy covers the full area of the surface, but does so at four times the resolution of the layer above, using four times the number of nodes as the layer above.

Rendering proceeds much like ROAM. The same $O(\log n)$ visibility optimization applies, and active quad-tree nodes are selected based upon the geometry resolution they provide, scaled by their distance from the view point.

Unlike ROAM however, frame coherence combined with the coarse granularity of the geometry leads to a relatively static working set. This allows batches to be stored in VRAM, to be reused rather than respecified at each frame. When an update becomes necessary, a batch is uploaded in a single large transfer, maximizing bus efficiency.

The relative infrequency of batch upload allows data access to be extended downward. A batch not found in RAM may be loaded from disk, giving a true out-of-core data access mechanism. Geomipmapping algorithms gain significant scalability in this fashion.

The basic geomipmapping algorithm given by de Boer [4] defines each node as a uniform grid of geometry, just as described in Section 2.1. A refinement of this technique, called "chunked LOD" by Ulrich [39],

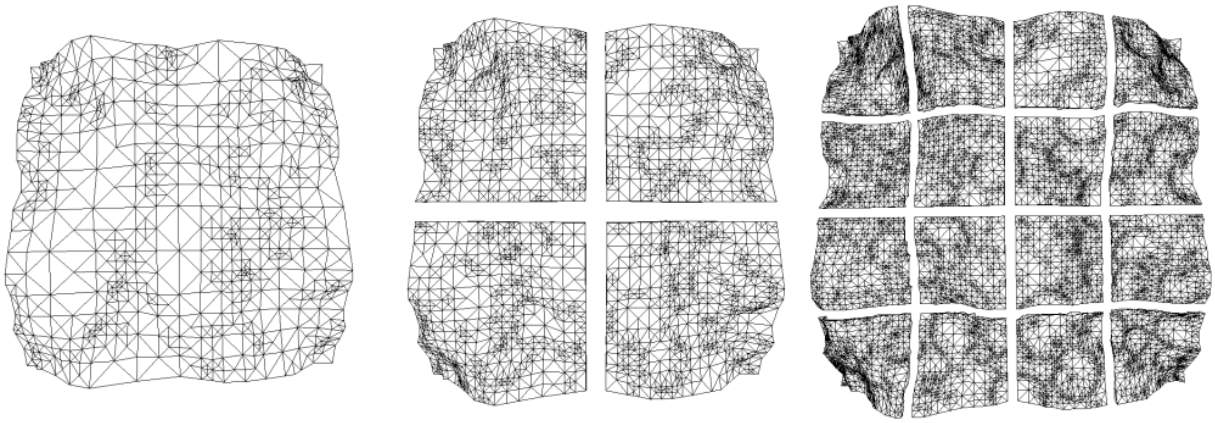


Figure 2.10: The top three levels of a chunked LOD tree, a variation on geomipmapping. Each successive layer covers the same area at 4× resolution.

optimizes the geometry using a bintree approach as in Section 2.3. Both gain the efficiency of static batching, though "chunking" does imply a preprocess.

2.4.3 Caveat: seams and skirts

All geomipmapping algorithms encounter the same T-intersection challenge described in Section 2.3.3. However, with coarsely-grained nodes, solutions lack the elegance of ROAM. In Figure 2.11 we see part of two adjacent geomipmap nodes, with the T-intersections marked.

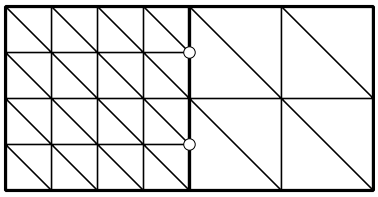


Figure 2.11: Adjacent geomipmap nodes with erroneous T-intersections.

In the case of basic geomipmapping [4] a clean solution uses multiple distinct triangulations of a given set of surface vertices. In Figure 2.12 we see a *seam* triangulation that avoids the problem vertices. While this solution requires a separate seam triangulation for each of the 4 edges of the node, it can be made very efficient using vertex indices to avoid duplicating geometry.

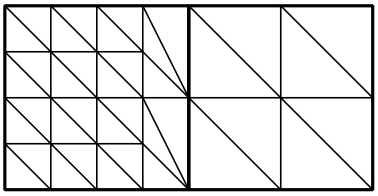


Figure 2.12: Adjacent geomipmap nodes with proper seaming, correcting T-intersections.

The chunked LOD [39] implementation must show extra caution, considering T-intersections in the preprocess, as apparent in Figure 2.10,

Finally, a brute-force solution exists. Gap-filling *skirt* triangles may be generated at each level-of-detail transition. Long, thin skirt triangles fill the type of gap depicted in Figure 2.8a, and degenerate (zero-area) skirt triangles eliminate edge sparkling in cases similar to Figure 2.8b. Google Earth [15] takes this approach.

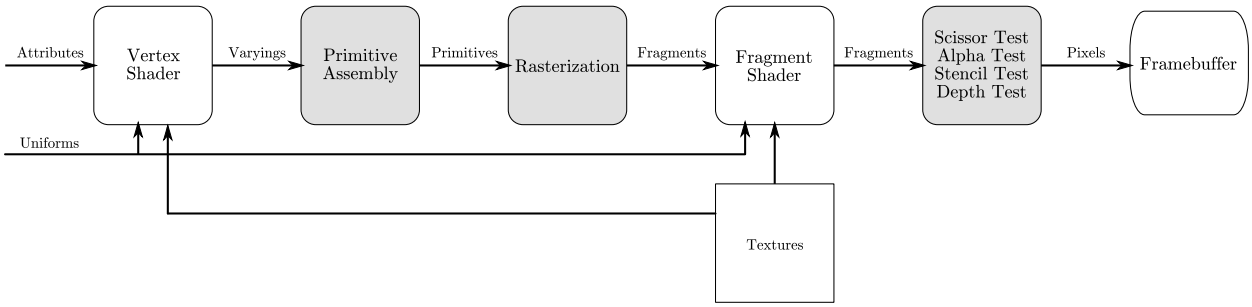


Figure 2.13: The OpenGL programmable pipeline. Contrast Figure 2.1. The vertex and fragment shader phases are new. (Gray areas are fixed; white areas are user-controlled.)

2.5 Geometry clipmaps

2.5.1 Motivation

Hardware capability continued to increase, as did the flexibility of the APIs used to control it. In 2001, with the release of the NVIDIA GeForce3 [28], hardware capability had reached a point where the complexity of the necessary APIs became intractable. Traditional CPU code would no longer suffice to comprehensively control the graphics hardware, and tools emerged allowing the programmer to develop processes to run on the GPU itself. Just as the movement of *data* toward the GPU transformed real-time thinking (as described in Section 2.4.1), so too would the movement of *instructions* toward the GPU.

Figure 2.13 depicts the structure of this new type of GPU. Contrast this with the fixed-function pipeline in Figure 2.1. Two areas have changed. The vertex transform and lighting stage has been replaced by the *vertex shader*, and the fragment texture/color/fog stage has been replaced by the *fragment shader*. These *programmable* pipeline stages are the points at which applications may inject custom processing in the form of "shaders," scripts written in GPU language. The vertex shader executes for each incoming vertex, and the fragment shader executes for each outgoing fragment.

To some extent, custom vertex shaders still do transform and lighting, and custom fragment shaders still do texturing, but applications are free to do as little or as much of these as necessary. Beyond emulating previously fixed functionality, new illumination models and visual effects become possible. Shadowing, surface relief, volumetric lighting, and skeletal animation are just a few of the techniques to become common given this flexibility.

Early shader dialects resembled assembly language, though high level languages soon followed. Support was fragmentary early on, with NVIDIA's Cg [25] working only with NVIDIA hardware and Microsoft's High Level Shading Language (HLSL) [26] working only with DirectX under Windows. In time however, the OpenGL Shading Language (GLSL) [18] arrived as a cross-platform standard, enabling GPU programming across all hardware and under any operating system.

A generalization of the input to the pipeline accompanies the generalization of its function. Previously, applications were required to submit specific types of geometric data: vertices, normals, texture coordinates, etc. These have been replaced by *attributes*, generalized vector values with meaning assigned by the application. The output of the vertex stage, which is the input to the rasterizer, has also been abstracted. Where the fixed-function rasterizer computed the position, color, and texture coordinate of each fragment, the programmable pipeline allows the

definition of arbitrary *varying* variables to be interpolated across each triangle. These varying variables are the primary means of communication from the vertex shader to the fragment shader. Finally, *uniform* storage allows the application to express constant data to be used as input to both the vertex and fragment stages.

This flexibility has triggered an avalanche of new ideas and techniques in the real-time computer graphics world. New literature continues to proliferate, and results published decades ago as photo-real techniques have been revisited and reformulated for real-time.

2.5.2 Algorithm

The *geometry clipmap* algorithm by Losasso and Hoppe [24] takes advantage of much of this new GPU functionality. At its core, a geometry clipmap is similar to a geomipmap. It is a hierarchy of batches, each representing a similar amount of geometry data, and each layer covering four times the area of the layer above at a quarter the resolution.

However, a geometry clipmap is more dynamic then a geomipmap. A geometry clipmap pyramid remains centered at the view point, moving with it, with VRAM geometry minimally updated with each change of the view (Figure 2.14).

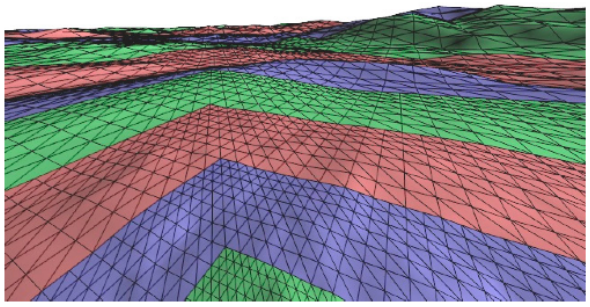


Figure 2.14: A geometry clipmap: geomipmaps centered on the view point giving small triangles near the viewer and large triangles in the distance.

To support these updates efficiently, the VRAM vertex buffers are accessed *toroidally*. While these buffers are fundamentally two-dimensional in layout, as is a common regular grid (Section 2.1), they wrap around along both axes. The top, bottom, left, and right edges do not necessarily coincide with the boundaries of the buffer or the stride of its rows, but instead are dynamic. Mapping such a data buffer is non-trivial, requiring offsets and modular arithmetic not provided by the fixed-function pipeline. This necessitates the use of the programmable pipeline.

As the view point moves, the data shift beneath it. Each change in the view discards some number of rows and columns as they move out of range, and requires the loading of an equal number of rows and columns as they come into view. It would be inefficient to actually move the data within the buffer, so only the *logical* origin of torroidal the buffer is moved. The new data are loaded into the same buffer locations vacated by the discarded data.

When such a move occurs, part of the triangulation of the vertices is invalidated. To repair this damage, the vertex indices of the triangulation are recomputed by the CPU and uploaded to VRAM. This is expensive and inelegant, and a later refinement by Asirvatham and Hoppe [1] resolves the problem using vertex shader texture access. In this formulation, the x and y values of the vertices are *constant* in eye space, and the z values are dynamically referenced from the height map, bound as a texture.

The inspiration for this toroidal update is acknowledged by the name. "Clipmap" refers to a feature of SGI's OpenGL Performer [6]. Clip texturing was a hardware mechanism that dynamically paged very large texture images from disk, usually based on view position. This mechanism used toroidal access to efficiently pan the VRAM image cache. Geometry clipmapping is a natural extension.

Note the similarity between the geometry clipmap output (Figure 2.14) and the geomipmap T-intersection flaw (Figure 2.11). The geometry clipmap algorithm makes no significant advance in the realm of T-intersection mitigation, and all of the potential solutions described in Section 2.4.3 apply. Losasso's implementation [24] does use vertex shading to perform a simple morphing at level-of-detail transitions, but must rely upon degenerate triangle

skirts to eliminate the remaining edge sparkle. Transitions in texture level-of-detail are blended away using fragment shading.

Losasso and Hoppe go to great pains to describe a land-form data compression method, and they do demonstrate browsing a 1-arc-second data set of the entire continental US in real-time. Despite this however, their implementation works strictly with in-core data. For this reason, the geometry clipmap algorithm does not scale. This restriction is slightly surprising given the authors' adoption of the term "clipmap," which initially referred specifically to an out-of-core mechanism.

2.6 Current hardware

2.6.1 GPGPU

The capabilities of real-time 3D graphics hardware continue to tend toward generality. Among the most significant recent enhancements has been an increase in available frame buffer formats. While frame buffers were previously limited to 32-bit pixels with each channel an 8-bit unsigned byte, current hardware provides pixels as wide as 128 bits with each channel a 32-bit IEEE floating point value.

Given the ability to both read from and write to such buffers, the practice of General Purpose GPU programming (GPGPU) has emerged. With as many as 128 4-channel vector parallel stream processors (on the NVIDIA GeForce 8800 Ultra [29]) total computational throughput approaches a teraFLOP in a single workstation PC. Developers were quick to adapt the existing practices of textured polygon rendering and programmable fragment shading to perform arbitrary computation, having no relation to the creation of 3D scenes.

Software soon evolved to support this practice. NVIDIA's Compute Unified Device Architecture (CUDA) [30] and Stanford University's BrookGPU [21] abandon the traditional concepts of vertices and fragments to abstract the hardware's processing capabilities. They provide languages and APIs that map more straightforwardly onto the variety of problems faced by users of high performance computing. Physical simulation, ray tracing, data encoding and encryption, computer vision, and image processing are just of few of the many fields that have benefited.

2.6.2 Geometry generation

Meanwhile, back in the world of 3D rendering, flexibility has also increased. New capability focuses on the *generation* of geometry. Just as the vertex and fragment phases of the 3D pipeline were made programmable in the last significant revision (Figure 2.13), now the primitive assembly phase has been made programmable (Figure 2.15).

This *geometry shading* phase allows GPU code to modify the topology of incoming geometry. Geometry shaders take incoming points, lines, and triangles and generate zero or more primitives, optionally routing them to one of several target frame buffers. "Transform feedback" allows this generated geometry to be sent back to VRAM output buffers, to be processed or rendered later.

A similar functionality allows floating point frame buffers to be bound and rendered as vertex buffers. Rather than drawing a distinction between color (r, g, b, a) and vector (x, y, z, w), buffer contents are treated simply as data. By relaxing the restrictions on the *meaning* of the values stored in VRAM, pixels may be used as vertices, taking advantage of pixel processing capacity in the processing of geometry.

2.7 Realism

There are a number of simple effects that may be applied to all forms of terrain rendering to heighten the level of realism in the final image. Here we discuss two of them.

2.7.1 Normal mapping

The first of these is known as *normal mapping*. During rendering, the illumination computation uses the normal vector of the surface, the vector toward the light source, and the vector toward the view point to calculate the reflectance of the light, and thus the

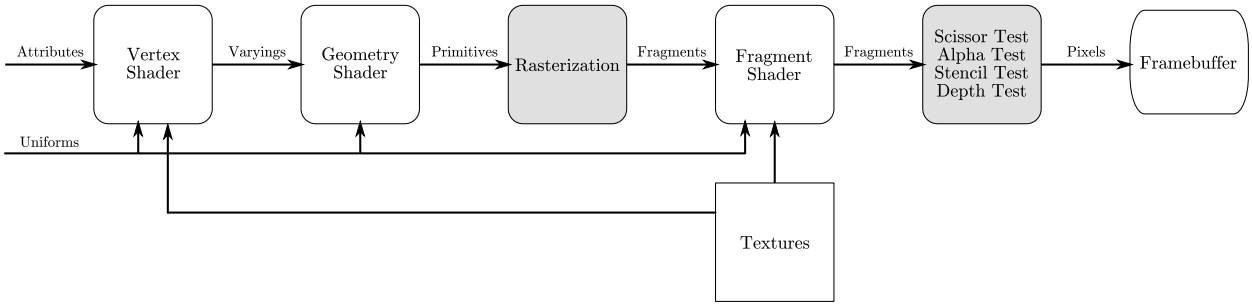
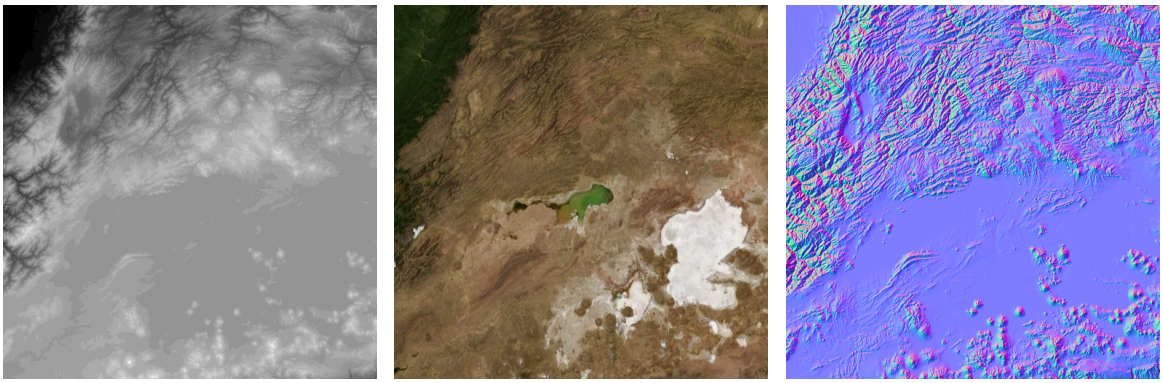


Figure 2.15: OpenGL geometry shader pipeline. Contrast Figure 2.13. The geometry shader phase is new. (Gray areas are fixed; white areas are user-controlled.)



(a) Height (b) Color (c) Normal

Figure 2.16: A height map (SRTM), color map (BMNG), and normal map (SRTM), each showing the same region.

appearance of the object. The resulting shading is a strong indication of the nature of the surface.

Traditionally, the normal vector of a surface is determined by the surface geometry, and is specified once for each vertex or triangle. However, if the normal vector is given instead by a texture map, then the illumination calculation may be performed on a per-pixel basis, rather than a per-vertex basis. This high-resolution shading gives the illusion of a great deal of geometric complexity, regardless of the true number of vertices and triangles. The basic concept of this dates back to the work of Jim Blinn in 1978 [2] and was known as "bump mapping" prior to the adaptation of the technique to modern real-time hardware.

In the case of uniformly-projected terrain, a normal map may be easily computed from a height map using a Sobel filter [37]. The result of this is shown in Figure 2.16. The components of a normalized vector fall in the range $[-1, +1]$, so to store a normal in a color map it must be offset and biased to the range $[0, 1]$. In this space the Z axis is $(0.5, 0.5, 1.0)$. For this reason, terrain (which usually faces up) produces roughly blue normal maps.

Real-time rendering using normal maps requires the use of fragment shading to shift the normal back to its native range, transform it into the local coordinate system tangent to the surface, and perform the illumination calculation on a per-fragment basis. The result of this process can be seen at the center of Figure 2.17, in contrast with the top of that figure, which lacks normal mapping.

2.7.2 Atmosphere rendering

Earth's atmosphere has a profound effect on the appearance of its terrain. Light passing through the atmosphere is randomly scattered and attenuated at various wavelengths due to absorption by air molecules and dust particles, varying exponentially with altitude. Light is scattered both in and out along the line of sight between any two points. This results in the blue of the sky, the yellow of a sunset, and the desaturation of the color of objects seen at a distance, known as "aerial perspective."

Earth data such as NASA's Blue Marble Next Generation [27] have had the contribution of the atmosphere subtracted. So for realism, it is necessary to add it back in. True simulation of atmospheric scattering is intractable, but a number of approximations have appeared. The most effective real-time algorithm developed to date is Sean O'Neil's GPU-based single-scattering approach [32].

The simulation of in-scattering and out-scattering involves the evaluation of nested line integrals. O'Neil's implementation is iterative, using curve-fit approximation functions to eliminate the inner integral, but necessarily stepping down the path of the light to evaluate the outer integral.

This is a relatively expensive operation to perform, but the visual impact of it is striking. Contrast the center of Figure 2.17 with the bottom. The output can be difficult to distinguish from a photograph taken from space.

2.8 Terrain on the sphere

All mention of scalability thus far has been with regard to the *quantity* of data, and scalability solutions focus upon data caching protocols. However, the scalability of the *extent* of data raises new issues. All examples of real terrain are spherical and as extent broadens, the underlying shape of a landform diverges from the simple plane assumed by all algorithms discussed to this point. Extending these algorithms to map from the plane to the sphere is relatively straightforward, involving basic trigonometry, and such an approach is common. Notably, Clasen and Hege adapt geometry clipmaps to the sphere[3], but at significant vertex shading expense. New problems arise, and achieving truly efficient rendering of terrain data on a planetary scale requires more careful analysis.

2.8.1 Spherical projection

Geographic data are commonly laid out following the familiar longitude/latitude grid known as a *cylindrical equal-area* projection (Figure 2.19). Its equally-sized rectangles map easily onto any of the rendering algorithms discussed previously. When mapped onto

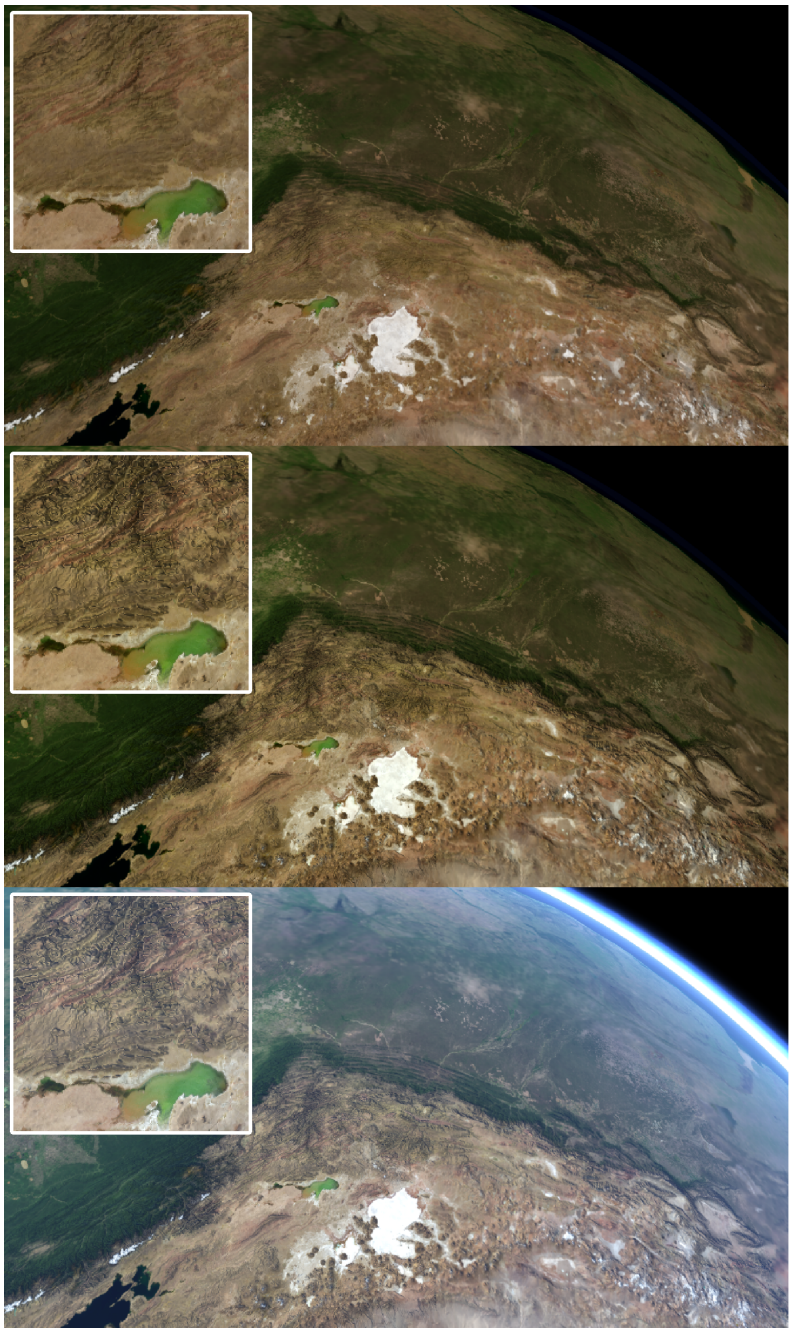
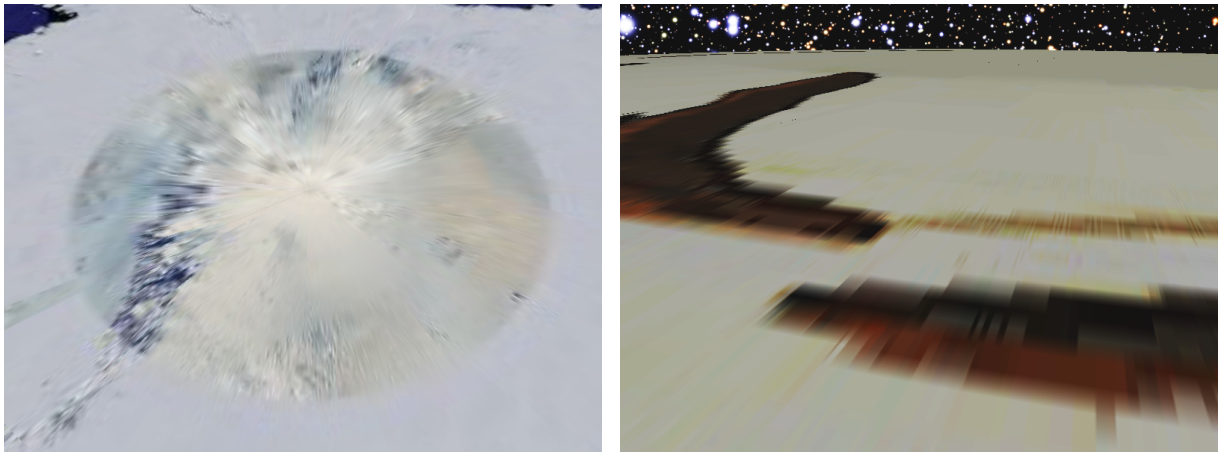


Figure 2.17: Effects for enhanced realism: basic illumination (top), per-pixel illumination (center), atmospheric scattering simulation (bottom).



(a) Google Earth (b) Mars Transporter

Figure 2.18: Polar distortion due to spherical projection: the stretched pixels caused by non-uniform data sampling in Antarctica (a) and at the north pole of Mars (b).

the sphere, the result resembles the common globe (Figure 2.20).

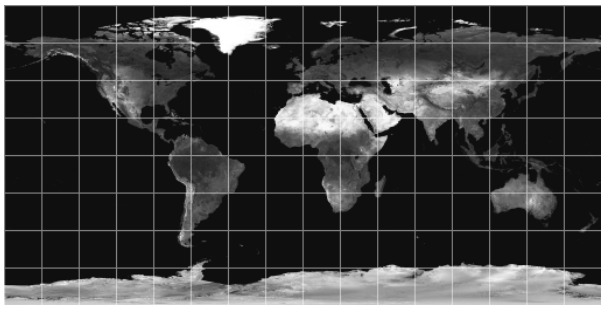


Figure 2.19: Earth longitude and latitude

While simple, this projection leads to a very non-uniform tessellation. All lines of longitude converge at the poles, and the width of each rectangle reduces to zero there. However, the width of each corresponding rectangle of data is a constant. The result is an over-sampling of data and geometry toward the top and bottom of the data set. Near the poles, data elements are squeezed longitudinally while retaining their height latitudinally, and the triangles of a spherical tessellation are compressed to zero area.

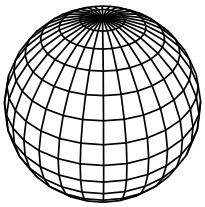


Figure 2.20: Spherical projection: parallel lines of latitude and lines of longitude meeting at the poles.

This over-sampling wastes storage and I/O bandwidth, and the visual impact of the anisotropic scaling is difficult to overlook. The effect is that of a radial blur. Note the distortion in Google Earth's [15] depiction of Antarctica in Figure 2.18a. The output fails to accurately represent the land-form, and zooming the view only magnifies the error. This flaw is nearly universal among planetary data rendering applications. Mars Transporter also suffers from it (Figure 2.18b) and this flaw provided some impetus to the pursuit of this research.

The solution to the problem is to generalize beyond the spherical projection, both in the layout of the data and tessellation of the sphere.

2.8.2 Stereographic polar projection

Spherically projected data *are* applicable in the region near the equator, and the majority of NASA and USGS data are made available in this form. Notably, the Shuttle Radar Topography Mission (SRTM) [8] provides useful data of the Earth within 60 degrees of the equator. The Mars Orbiter Laser Altimeter (MOLA) [47] data set extends to 88 degrees above and below the Martian equator, and fills the remaining gap using a *stereographic polar projection*. This projection, as depicted in Figure 2.21, is critical to reliable data sampling at the poles.

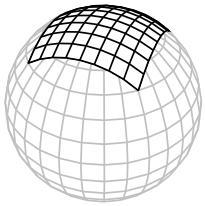


Figure 2.21: Polar projection: a uniform grid applied over the difficult area at the poles.

A stereographic polar projection is defined as a Cartesian coordinate system slicing the globe at a specific latitude. Just as spherically projected data are correctly scaled only at zero latitude (the equator), stereographic polar projected data are correctly scaled only at the defining latitude. Just as spherical data become unusable near the poles, polar data become unusable near the equator. Together the two provide useful coverage of the entire globe. The Landsat Image Mosaic of Antarctica (LIMA) [42] is made available entirely in such a stereographic polar projection.

2.8.3 Spherical tessellation

Where once we had uniform grids of data mapping onto uniform grids of geometry, we now have a situation where both data and geometry are non-uniformly sampled, and where their samplings do not necessarily coincide. This gives us the opportunity to choose a spherical tessellation more appropriate for the problem at hand.

There exist many spherical tessellations of the sphere that do not follow lines of longitude and latitude. The most useful tessellations are inexpensive to generate, refinable to arbitrarily granularity, and largely uniform.

The uniformity of a spherical tessellation can be judged visually, but also quantified. We want the degree (the number of incident edges) of all vertices to be similar, the inner angles of all faces to be similar, and the lengths of all edges to be similar. A uniform tessellation minimizes the variation in all of these.

Common practice in spherical tessellation uses subdivision of regular polyhedra. Polyhedral subdivision proceeds recursively. Each face of the polyhedron is divided into similar sub-faces, and each vertex is normalized to fall on the surface of the unit sphere. This is recursively repeated for each sub-face, and the depth of recursion determines the final granularity. The nature of the initial polyhedron determines the uniformity of the final tessellation.

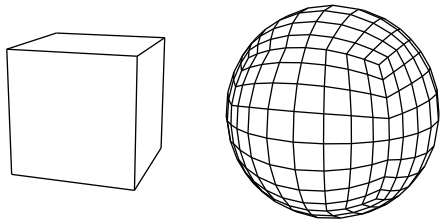


Figure 2.22: A recursively subdivided cube, a relatively poor tessellation of the sphere.

Figure 2.22 shows a recursively subdivided cube. At each step, square faces are subdivided into square quadrants. The original 8 vertices of the cube are still visible as *nodes* in the final tessellation. The nodes are the points where the tessellation is least consistent. Much like the poles of the standard spherical tessellation, this is where non-uniform sampling occurs. In the case of the cube, the nodes have degree

3, while the all other vertices have degree 4. The internal angles of the faces at the nodes are 120°, but tend toward 90° at the center of the original cube faces. Edge lengths on the subdivided cube are quite inconsistent.

Despite lackluster uniformity, the subdivided cube does result in a tessellation consisting entirely of quads, which is advantageous when considering planar terrain algorithms originally devised to work with rectangles. Sean O'Neil used the subdivided cube when adapting the ROAM algorithm to the sphere [31]. Hwa et al [16] did the same, and used the resulting quads to generate a tessellation of (nearly) right triangles, the basis for a unique texture tiling approach based upon 45-degree rotations.

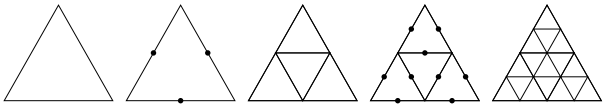


Figure 2.23: Triangle subdivision: bisect the edges and connect the vertices.

The next regular polyhedron is the octahedron, consisting of 8 equilateral triangles. To subdivide a triangular face, each edge is bisected and the resulting vertices connected, as in Figure 2.23. The octahedron and the result of three recursive subdivisions are shown in Figure 2.24. The nodes have degree 4 and an angle of 90°. Away from the nodes, degree is 6 and angles tend toward 60°. If the terrain level-of-detail algorithm does not rely upon quads, then better uniformity can be achieved using triangles in this fashion. Microsoft Research's Hierarchical Triangular Mesh uses this structure to index data on the celestial sphere [38].

A perfect tessellation would have degree 6 and 60° angles across its entire surface. Unfortunately, this is impossible (given the requirement that a tessellation be finite). The best that can be done is to begin with the most complex convex regular polyhedron, the icosahedron, with 20 equilateral faces.

Recursive subdivision of the icosahedron is shown in Figure 2.25. Vertex degree is 5 at the 12 nodes and 6 elsewhere. Angles vary from 72° to 60°. Edges are

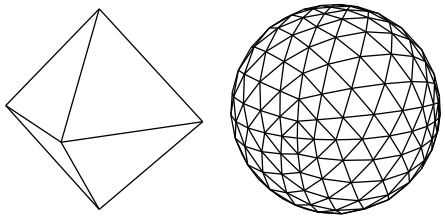


Figure 2.24: A recursively subdivided octahedron, a passable tessellation of the sphere.

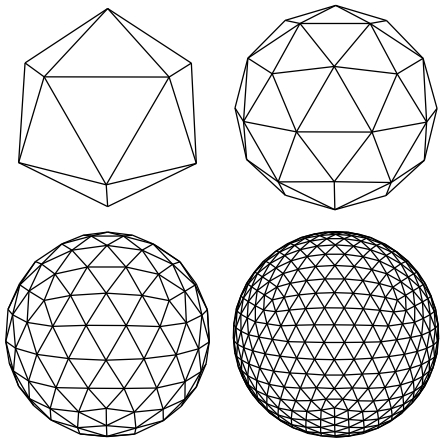


Figure 2.25: A recursively subdivided icosahedron giving a sphere with very uniform tessellation

nearly identical. This is excellent uniformity, and the nodes of the tessellation can be difficult to point out without close scrutiny. This construction is familiar to many 3D programmers, as it is the first non-trivial example in Chapter 2 of *The OpenGL Programming Guide* [46].

R. Buckminster Fuller based much of his work on the icosahedron. His geodesic dome is a cap of a recursively subdivided icosahedron. Fuller also set the historical precedent for mapping the earth using the icosahedron. In 1946 he proposed the Dymaxion™ Map as a method of cutting the planar projection of the world in such a way that the edges of the map fall over the oceans rather than the land. While his original design was based on the cuboctahedron, a

1954 refinement used the icosahedron, as seen in Figure 2.26 [11]. It is coincidence that Earth's continents fall within the faces of the icosahedron, but the Dymaxion™ Map is indicative of the uniformity gains of a projection having 12 poles instead of only two.

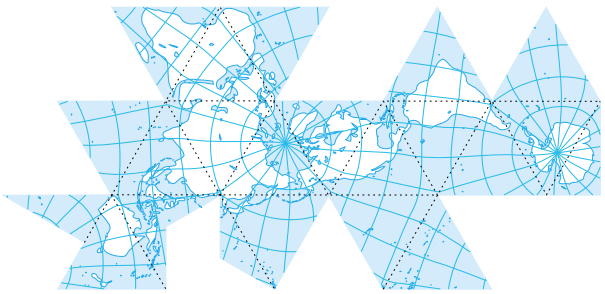


Figure 2.26: Dymaxion™ Map, fitting the continents within the faces of the icosahedron.

2.9 Moving forward

It hopefully clear from the discussion in this chapter that the history of terrain rendering is necessarily tied to the capabilities of the hardware available at the time. Terrain algorithms are often the first to take full advantage of these capabilities as they arise.

In review, table 2.1 summarizes the literature discussed in this chapter. Check-marks indicate whether each algorithm involves an extensive pre-process before rendering, a uniform triangulation, adaptive level-of-detail, out-of-core data access, significant CPU load, and the programmable GPU pipeline. Spherical adaptations are listed where they were found in the literature, though it should be noted that any algorithm can be mapped to the sphere given a proper spherical tessellation, usually cubic subdivision.

This table does not enumerate the concepts underlying terrain composition. As noted previously, terrain literature focuses exclusively on the careful tessellation of height maps and the basic application of registered color maps. Flexible approaches to the combination of disparate terrain data sets in a common visualization are not to be found. This is the gap to be filled by this research. It is a true opportunity to address real concerns in geodata visualization that have yet to be formalized.

This work will build upon everything that has been presented in this chapter, taking full advantage of the flexibility of modern hardware. We will see a new approach to planetary-scale land-form data management, geometry generation, and real-time display that has only recently been made possible by advances in hardware development.

Algorithm	Pre-process	Uniform	LOD	Out-of-core	CPU-heavy	GPU	Sphere
Regular Grid		✓					
TIN [12]	✓						
ROAM [5]	✓		✓		✓		[31]
Geomipmap [4]		✓	✓	✓			
Chucked LOD [39]	✓		✓	✓			
Geoclipmap [24]	✓	✓	✓		✓	✓	[3]

Table 2.1: The terrain of terrain literature

Chapter 3

Composition Algorithm

This chapter describes in detail a GPU-centric real-time approach to generating and rendering planetary bodies composed of arbitrary quantities and types of height-map data, textured and illuminated using arbitrary quantities and types of surface-map data. An implementation of this approach has demonstrated all of the concepts presented here.

At the core of this approach lies the assertion that modern graphics hardware need not draw a distinction between colors and vectors. As described in Section 2.6.1, commodity GPUs support both reading from and writing to 4-component 32-bit IEEE floating point image buffers, as well as the application of such image buffers as renderable geometry buffers. With the distinction between color (r, g, b, a) and vector (x, y, z, w) blurred, a variety of highly efficient image processing operations become applicable to both terrain geometry and terrain surface maps.

The algorithm itself is described here. A number of composition operations enabled by it are examined in Chapter 4. Finally, the performance characteristics of it are measured and presented in Chapter 5.

3.1 Overview

For each frame, the display of a planetary body proceeds in three major phases.

1. Visibility determination (Section 3.2)
2. Geometry generation (Section 3.3)
3. Rendering (Section 3.4)

These phases form the three stages of a pipeline, as shown in Figure 3.1. They are distinguished from one another by the explicit data transfer of the output of one to the input of the next. These data transfers may be performed in parallel with other processing. This leads to efficiency gains when rendering multiple bodies. For example, if both the Earth and the Moon are composed within a scene, then the visibility phase is executed for both, followed by the geometry generation phase of both, and finally the rendering of both. Visibility determination of the Moon proceeds while the the Earth's visibility solution is in transfer, Earth's geometry is generated while the Moon's visibility is in transfer, and so on. This reduces stalls in both the CPU and GPU, giving increased throughput.

The following sections describe each of these phases in detail. You'll note a constant battle with issues of numerical precision. Precision is among the focuses of this work. At a number of points, both a naive approach and a more considered approach are presented and contrasted.

3.2 Visibility

The visibility phase is executed by the CPU. The primary goal of this phase is to perform the initial subdivision of the icosahedron (Section 2.8.3) in order to seed the geometry generation phase. This subdivision is *not* the final spherical tessellation, it is merely a gross determination of visibility and granularity.

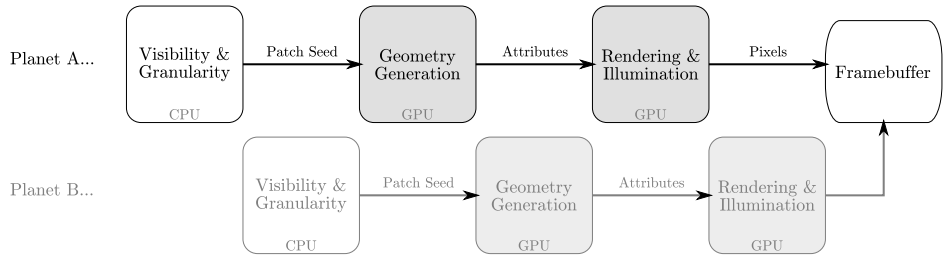


Figure 3.1: The planet rendering pipeline, showing the interleaving of processing and data transfer when displaying 2 planets

3.2.1 Patch enumeration

We begin with the basic icosahedron. Throughout the run time, we maintain a frame-coherent hierarchy giving the current faces of a subdivision of that icosahedron, referred to as surface *patches*. This hierarchy is a tree structure with a constant number of leaves. The management of this tree proceeds similarly to the ROAM algorithm (Section 2.3), but while ROAM is concerned with the maintenance of a continuous triangulation free of T-intersections, we are concerned only with a coarse patch triangulation. Thus, we have the luxury of ignoring T-intersections until after refinement (Section 3.4), and our hierarchy maintenance algorithm is simplified accordingly.

As the view varies from frame to frame, those patches 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 patches numbering as near as possible to (though not larger than) a constant n_p, and we maintain this set with a simple priority algorithm. Patches are sorted by the solid angle that each subtends from the current view. If the set of patches is too small, the largest patch is subdivided, as in Figure 2.23. If the set of patches is too large, the smallest set of four sibling patches is collapsed. The result is a coarse set of approximately n_p visible patches. These appear as in Figure 3.4.

In the circumstance where the target display is stereoscopic or tiled, the view volume may consist of multiple overlapping or disjoint frusta. A patch passes if it falls within any part of the view volume, thus the visibility phase need be executed only once regardless of the number of view points or view ports.

This process is equivalent to the visibility test of the geomipmapping algorithm given in Section 2.4.2. In both cases, it is a coarse visibility test, selecting potentially visible geometry batches rather than culling on a per-triangle basis.

3.2.2 Horizon

In addition to the culling planes defining the view volume, a *horizon* plane enables additional patch culling opportunities. This plane is independent of the structure and orientation of the view volume, depending only on the view point and planet. Figure 3.2 shows the cone of occlusion of the user's view, tangent to the planet's surface. The horizon plane is computed using the right triangle defined by the planet's radius r and the viewer's distance from the center of the planet d. The horizon plane normal is the normalized vector from the planet center to the view point, and the horizon plane distance from the center of the planet is r^2/d. Any patch behind this plane falls within the planet's occlusion cone. As d approaches r the horizon plane culls the majority of the sphere. In practice, the horizon is the most effective, and thus first-tested, culling plane.

3.2.3 Patch bounds

Testing a patch for visibility is more complex than simply checking a triangle against culling planes. The surface of the planet within the triangle's bounds is not planar. Terrain data will eventually be mapped

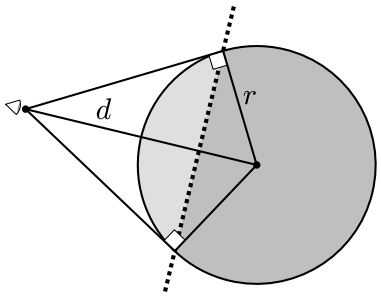


Figure 3.2: The horizon plane. The viewer at distance d from the center of a planet of radius r, cannot see the dark area.

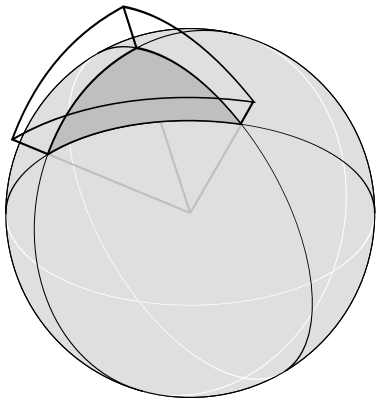


Figure 3.3: A triangular surface shell, a terrain bounding volume defined by three planes and two radii

within that area, and we must determine whether the geometry of this terrain is or is not visible.

As shown in Figure 3.3, the three sides of a triangular area on a sphere are segments of geodesics ("great circles" cutting the sphere into equal halves). Let these geodesics define three planes cutting through the center of the sphere. These planes form a wedge-shaped volume. The terrain within this volume has some minimum and maximum altitude, which give radius extrema. The three planes and two radii define a surface *shell*, a tight bound on the terrain within the patch. We may determine the visibility of the terrain within the patch by testing the patch's bounding shell.

3.2.4 Output

The final result is a depth-first traversal of all of the visible patches tessellating the sphere. Rendered, they appear as Figure 3.4. These patches are uploaded to VRAM as the initial "seed" input to the GPU process that generates land-form geometry. The number of patches n_p will determine the number of rendered batches, so its value may be set around 300, as recommended in Section 2.4.1.

This upload is performed by loading the three vectors defining the corners of each patch into 2D floating point texture maps, as suggested in Section 2.6.2. There are three such texture maps, one storing the position of the corner, another storing its normal, and a third storing its spherical latitude and longitude. The layout of these is shown in Figure 3.5. These buffers are asynchronously transferred using the OpenGL pixel buffer object extension [19].

3.2.5 Caveat: precision

The mean radius of the Earth is 6,372,797 meters. Unfortunately, a 32-bit IEEE floating point value provides only around 7 digits of precision [17]. A modern GPU works with 32-bit floats internally, so the processing of Earth terrain occurs at or near the limit of the computational precision of the GPU.

This means that surface features on the scale of a meter cannot be reliably represented in a coordinate system with its origin placed (quite reasonably) at the center of the planet. To do so results in geometry dominated by numerical precision artifacts on one-meter scales. However, there is a clear motive to enable the presentation of detail on the scale of a meter and below, as this is the scale of human experience.

To resolve this problem, patches are generated not in object space (the coordinate system of the planet), but in eye space (the coordinate system of the viewer). This coordinate system has its origin at the user's eye, with the X axis pointing to his right

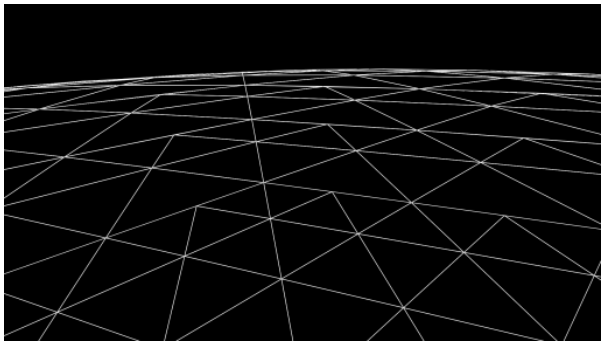


Figure 3.4: The output of the visibility/granularity phase, a rough triangulation of the visible portion of the sphere

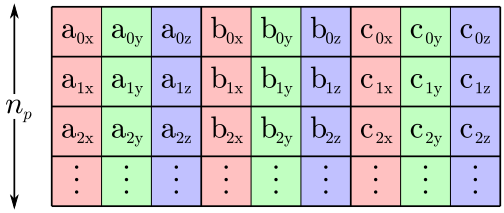


Figure 3.5: The geometry generation seed of n_p patches, each with 3 vertices $[a, b, c]$, encoded as a $3 \times n_p$ RGB texture

and the Y axis pointing up. As the user navigates the universe, he remains motionless and the universe moves and rotates about him.

The advantage of this follows from the nature of an IEEE floating point variable, and the distribution of the possible values that it may take. Floats concentrate their effectiveness near zero, and the granularity of representable small values is high. A float *can* represent very large values, but the gaps between representable large values are wide.

So, by generating the geometry of planets in eye space, we gain the ability to represent land-forms with high precision near the user, relative to his scale. Distant landforms are necessarily generated with lower precision, but the quantization is too far away to be visible as error.

Complicating this is the assumption that the user is in constant motion due to his interaction with the scene. Eye-space is always changing. Thus, generated planetary geometry must be regenerated at each rendered frame.

It would seem ridiculous to ignore the advantages of a frame-coherent tessellation, but experience has shown that it is not. The geometry generation algorithm presented in the next section has little impact on run-time performance. Because geometry generation occurs in VRAM, and is performed entirely by the GPU, the full potential of GPU stream processing is brought to bear. Modern 3D hardware is more than capable of discarding and regenerating the entire scene with each new frame.

3.3 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 land-forms, with optimal level-of-detail.

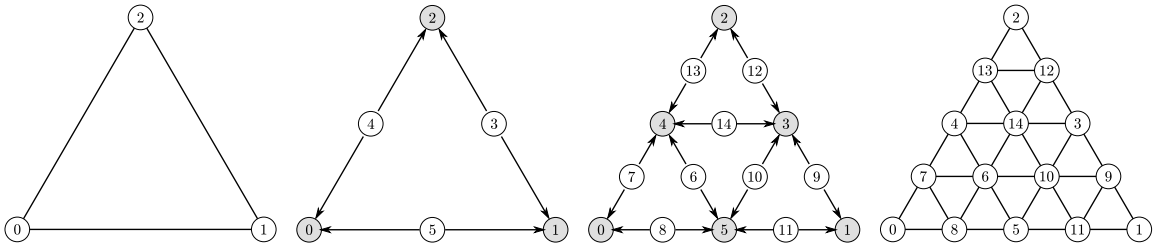
The input, described in the previous section, gives a low-resolution triangulation of the smooth sphere, with each triangle of roughly the same *visual* size in eye space. So, the task of the geometry generation phase is to perform a subdivision of the input mesh, and a displacement of this subdivision's vertices using height data.

Geometry generation is a GPGPU process, as described in Section 2.6.1. Floating point textures act as vector data buffers, and GLSL shaders operate upon these, writing their output to color render targets to be used in subsequent operations.

3.3.1 Subdivision

Triangle subdivision is generally a recursive process, as described in Section 2.8.3. To map this process onto the GPU, we must formulate it iteratively. To do this, first note that the number of vertices n_v produced by the subdivision of a triangle to recursive depth d is

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



(a) Input (b) Depth 1 (c) Depth 2 (d) Output

Figure 3.6: Vertex indices and intermediate subdivision steps for a recursion of depth 2

Here are the vertex counts for several small values of d, commonly used when doing real-time subdivision.

d	$n_v(d)$
0	3
1	6
2	15
3	45
4	153
5	561
6	2145
7	8385

For clarity, the figures in this section show a subdivision depth of 2, with 15 vertices per patch. In practice, the depth is usually 4 or 5. This value adjusts the batch granularity as described in Section 2.4.1. A high value results in fewer patches emitted by the visibility phase, larger batches, and less-effective view culling. A small value places a greater load on the CPU but results in precise visibility. This allows the balance of the algorithm to be tuned to the hardware. Older hardware, such as the NVIDIA GeForce FX or ATI Radeon 8500 run smoothly at a depth of 3, while a GeForce 8800 can manage at 6. As we will see, n_v translates into a texture buffer width, which has a maximum of 4096 on recent hardware, and 8192 on current hardware, giving an upper bound on the selection of d.

With the exception of the three initial vertices defining a triangle (v_0, 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. Figure 3.6 depicts this relationship graphically, with arrows indicating dependence.

At start-up, the j and k indices for each i are generated using such a breadth-first traversal and stored in a constant look-up table, as in Figure 3.7. This table is stored in VRAM as a 2-channel 16-bit texture, thus encoding the recursive subdivision relationship in a form easily accessible by a GLSL shader.

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	...

Figure 3.7: Vertex index dependence look-up table, encoding the relationship depicted in Figure 3.6.

3.3.2 Iteration

Now begins the iterative process of subdivision. The initial input is the texture buffer depicted in Figure 3.5, with n_p rows, one row per patch. The output is a similar texture buffer, but with the number of columns expanded from three to accommodate the final vertex count n_v.

Patches are processed in parallel. Each step of the iteration computes one level of depth d, adding $n_v(d) - n_v(d-1)$ vertices (columns) to the output for each patch (row). The process is depicted in Fig-

ure 3.8. At step i the output texture is bound as render target and a rectangle is drawn from pixel $[n_v(i-1), 0]$ to pixel $[n_v(i), n_p]$, causing the GLSL fragment shader to execute for each of the new vertices being generated.

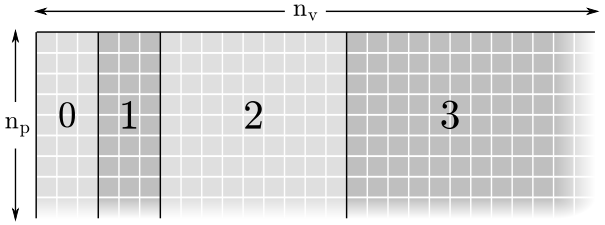


Figure 3.8: The vertices calculated at each depth step of the iterative parallel subdivision process

The fragment shader looks to its fragment coordinate to know which vertex index it is generating. It refers to the index dependence look-up table to determine the indices of the vertices this depends upon, and it uses these values as texture coordinates to look up the input vertices themselves in the input texture. It then performs its computation (described below) and writes the resulting vertex to its fragment in the output texture.

Note, iterative GPGPU processes must often utilize a technique known as "ping-ponging". A GPU cannot both read from and write to a single buffer simultaneously, as to do so would introduce data dependencies that undermine parallelism. So in circumstances where the output of a computation should overwrite its input, a pair of buffers A and B is used. One iteration reads from buffer A and writes to buffer B, the next iteration reads from B and writes to A, and so on. The iterative subdivision process is an example of this.

As noted, there three distinct subdivided attributes: vertex position $\bar{p}$, normal $\bar{n}$, and spherical coordinate (θ, ϕ). These attributes are computed as follows.

3.3.3 Normal computation

Normal subdivision is performed as expected.

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

However, a trick is introduced. The output image is a 4-channel floating point 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.

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

3.3.4 Lat/lon computation

Under ideal circumstances, it would not be necessary to compute the latitude and longitude coordinates of each vertex by recursive subdivision. Normally, one would not even bother to store spherical coordinates, preferring to simply compute them from the vertex normal at render time:

$$\begin{aligned} \phi &= asin(\bar{n}_y) \\ \theta &= atan2(-\bar{n}_z, \bar{n}_x) \end{aligned}$$

However, this method is imprecise and unstable. Generating spherical coordinates in this fashion leads to 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. We can do much better using the haversine geodesic midpoint method to subdivide the coordinates of the input triangles.

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

While more expensive, this method produces reliable spherical coordinates even at sub-meter scales. It also works correctly with coordinates outside of the range $\pm\pi$, which greatly simplifies mapping data across the International Date Line.

3.3.5 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.

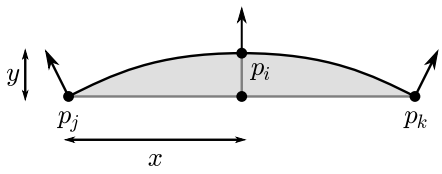


Figure 3.9: Precise position subdivision without reference to radius

See Figure 3.9. We take the angle between the input normal vectors (as found during normal computation) and compute its tangent using a constant look-up 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° with little noise. Thus, the table look-up 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.

3.3.6 Displacement

Height maps are naturally images. Given that the surface of the planet is now represented in VRAM as an image, surface displacement is simple image composition, an operation the GPU is most adapted to perform efficiently.

The newly-generated position, normal and spherical coordinate buffers are bound for reading. An $n_v \times n_p$ position accumulation buffer is bound as render target. Height maps are loaded as textures, usually formatted as 16-bit single-channel luminance. They are bound one-by-one, and a render pass for each is made. A rectangle covering the accumulation buffer is drawn, triggering the execution of a GLSL shader for each fragment.

At each fragment, the height map texture coordinate must first be determined. For spherically projected data, this is the value in the spherical coordinate buffer. For polar projected data, this is the value in the normal buffer, offset and biased as per the projection.

In addition to the texture coordinate, a texture quality value may be computed comparing the effective resolution of the projected height map data versus the resolution of the tessellation at the current vertex. A significant mismatch indicates badly projected data, such as a vertex near the pole of a spherical projection, or near the equator of a polar projection. This gives a weighting factor used to penalize bad data. This will be discussed in greater detail in Section 4.1.

Once the texture coordinate and quality are known, the height map is sampled. The displaced position of the vertex is the base sphere position plus the normal scaled by the height value. The quality factor gives the weighted average of this displaced position and the current accumulated position, resulting in a refined accumulated position. As some GPUs are incapable of blending floating point render targets, this process ping-pongs a pair of accumulation buffers, as described above.

The fundamental effect of this process is that the input height map data, regardless of projection and resolution, are naturally re-sampled to the optimal tessellation mesh computed previously. Overlapping height maps are blended. Under-sampled height maps are interpolated by the GPU's linear magnification filtering hardware. Over-sampled height maps are down-sampled by the linear minification

hardware. Any discontinuity between superimposed height maps of different resolution are obscured by being re-sampled to a common mesh. And any low-quality height data at the extremes of the projection are automatically weighted away in deference to the more appropriately projected height data composed with it.

This 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 of Section 5.1.4.

This approach to height map accumulation affords an opportunity to satisfy the last criterion for continuity. Lindstrom et al [23] enumerate three aspects of continuity of terrain level-of-detail under view point motion. In summary, they are (i) geometry morphs between discrete levels of detail instead of popping, (ii) adjacent blocks of geometry align without gaps, and (iii) the number of triangles tessellating a given area varies smoothly. In our method, (iii) is satisfied during the visibility and granularity phase (Section 3.2), and (ii) is satisfied during rasterization (Section 3.4), but (i) remains.

However with this 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 built-in 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 detail coefficient, as used in existing approaches to geomorphing [9]. At the time of this writing, this advanced terrain sampling operation is untested and the current implementation does reveal popping artifacts. Recognizing the necessity of continuous LOD in any modern approach to terrain rendering, we must place trilinear filtered geometry among the most important areas for future work.

3.3.7 Output

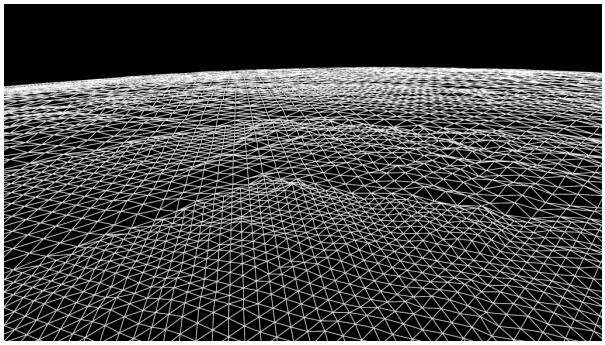


Figure 3.10: The output of the geometry generation phase, a fine triangulation with height displacement

We now have the final vertices of the geometry representation stored in a 32-bit floating point texture. In wire-frame, they appear as Figure 3.10. Contrast this with the input tessellation in Figure 3.4, showing the planet from the same view point.

In the next phase, we will wind these into triangles and rasterize them. Before we may do so, they must be moved from their image buffer to a vertex buffer. As before, this transfer is performed asynchronously within VRAM using the OpenGL pixel buffer object extension. The spherical coordinate buffer, normal buffer, and accumulated position buffer are concatenated onto a single vertex buffer suitable for rendering.

Note that the normal buffer does *not* represent the normals of the displaced vertices, but remains useful to us when later computing polar-projected texture coordinates.

3.3.8 Aliasing and Error

The height map displacement process does not enforce a constraint that the vertices of the source data fall upon the vertices of our triangulation. This is a significant departure from one of the basic assumptions made in the field of terrain rendering. As discussed in Chapter 2, most terrain rendering approaches utilize a regular grid of data applied to a reg-

ular geometric grid of right triangles, but the goal of supporting arbitrarily projected data precludes this. This has a two-fold impact. First, it allows aliasing to occur when data is resampled to our triangular mesh. Second, it adds complexity to the analysis and mitigation of error.

The analysis of error is a key aspect of terrain rendering research. Note that the refinement processes of both the triangulated irregular network approach of Section 2.2 and the ROAM approach of Section 2.3 are defined in terms of error mitigation.

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 non-uniformity of such mappings on aliasing and error. A thorough analysis of the error that may arise here is necessary before the technique can be embraced as a reliable tool. This analysis is left for future work.

3.4 Rendering

The first step toward rendering the vertices generated by the previous phase is to wind them into triangles. We immediately face the same T-intersection challenge described in Sections 2.3.3 and 2.4.3. In this case, there exists a straightforward approach to the elegant solution of Figure 2.12.

3.4.1 Patch winding

As with ROAM, we mandate that adjacent patches must be within one level in depth. So a patch of depth d has neighbors only of depth d, $d-1$, and $d+1$. When considering a patch of depth d, the $d+1$ neighbor case may be ignored because *this* patch is of depth $d-1$ relative to *that* patch, thus only edges adjoining d to $d-1$ need special treatment.

If a patch of depth d lies adjacent to a patch of depth $d-1$ then the adjacent edge must be wound at half resolution. A patch may lie adjacent to a coarser patch along zero, one, two, or three sides. The one and two-side cases have three possible orientations, for a total of eight possible edge windings. Figure 3.11 shows all necessary triangulations of a patch of depth three.

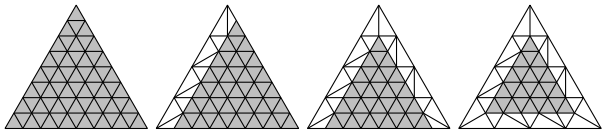


Figure 3.11: All possible windings (modulo rotation) of a patch of depth three adjacent to zero or more patches of depth two

The eight cases are enumerated and stored in VRAM using OpenGL element buffer objects. Element buffers are lists of indices referring to elements within a vertex buffer. Because the vertex buffer is bound independently from the element buffer, these eight element buffers may be used with all n_p patches, and the penalty for over-specifying elements is negligible.

Patch neighborhoods are implicit. During rendering, a context-aware depth-first traversal of the set of active patches is performed. The context of the traversal is the neighborhood of the current patch. As is common in tree traversal, if a patch's child reference is empty then that patch is a leaf to be rendered. In the context-aware traversal, if a patch's neighbor reference is empty then that patch is adjacent to a lower-resolution patch (the neighbor traversal has "fallen off the end" of the tree). The non-null neighbors are counted and the appropriate element buffer is selected from the eight. The vertex buffer and element buffer are bound and rendered.

3.4.2 Deferred texturing

As VRAM quantities have increased and off-screen rendering has become efficient, the tactic of *deferred shading* has become common. Originally developed under the name "G-buffering," [34] a deferred shading renderer uses a pair of off-screen render targets in a rendering pre-pass which rasterizes objects without illuminating them. One render target receives the diffuse color and the other receives the surface normal at each pixel.

Illumination is performed on-screen. The volume of influence of each light source is rendered, and a fragment shader executes for each pixel of that volume. The shader references the color and normal from the off-screen buffers, and accumulates the illuminated surface in the frame buffer.

The advantage of deferred shading is that the complexity of the scene is isolated from the complexity of the illumination. Given n objects and m light sources, traditional rendering requires that each object be rendered with each light source, which is $O(n \cdot m)$. The deferred approach renders all objects in the off-screen pass, and all light sources in the on-screen pass, which is $O(n+m)$, an efficiency gain for large m.

When composing disparate surface maps, it is advantageous to generalize this concept. We do not require many light sources per planet, we require many image maps, and we do not want to re-render our planet for each. So the initial *deferred texturing* pass begins by rendering the triangulated geometry of the planet and writing the texture coordinates themselves to an off-screen 4-channel 32-bit floating point render target as $(\theta, \phi, \bar{n}_x, \bar{n}_z)$. See Figure 3.12.

Note, this is the *only* time 3D geometry is drawn, and all other rendering is performed in screen space.

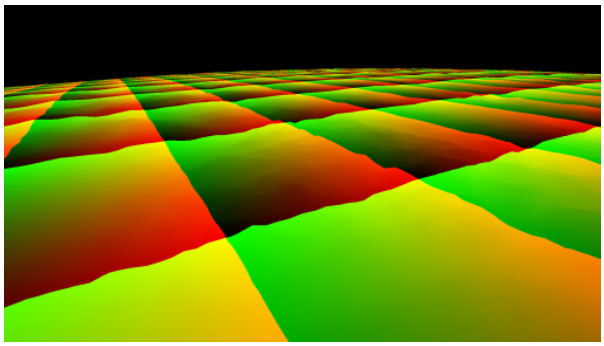


Figure 3.12: The texture coordinate buffer showing θ in red and ϕ in green (scaled by 100 for visibility) for each point on the surface of the planet.

3.4.3 Surface map accumulation

Just as height maps of arbitrary projection and type are adaptively composed as images during geometry displacement (Section 3.3.6), so too are surface maps composed during rendering.

The bounding volume of a given surface map is rendered to an off-screen buffer, triggering a shader at each fragment. For a given fragment, the texture coordinate is taken from the buffer produced in the previous step. Spherically projected surface maps derive their texture coordinate from θ and ϕ while polar projected surface maps use $\bar{n}_x$ and $\bar{n}_y$, computing $\bar{n}_y = \sqrt{1 - \bar{n}_x^2 - \bar{n}_z^2}$ if necessary.

In addition to these, the depth buffer gives the distance to the fragment, which may be combined with the view projection matrix to determine its eye-space 3D coordinate. This coordinate may be transformed in a variety of ways enabling, for example, a perspective-projected surface map such as a landscape photograph to be correctly and adaptively applied.

Texture quality is computed as before, and the surface map is biased and blended with the accumulation buffer. Figure 3.13 shows the accumulated normal maps of the scene shown in Figure 3.10. Figure 3.14 shows the accumulated diffuse color maps.

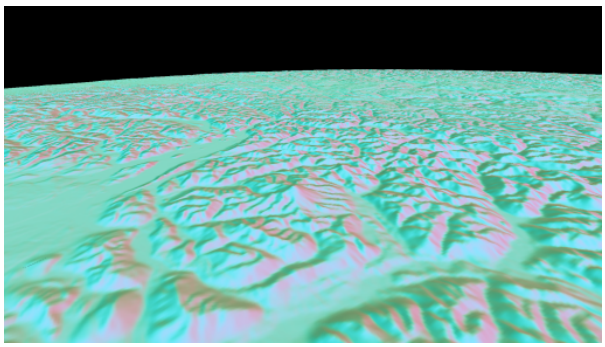


Figure 3.13: The normal accumulation buffer

There is limitless potential in this phase. It is here that color space transformations may be made, isolated channels from unregistered data sets may be combined, and time-varying data may be interpo-

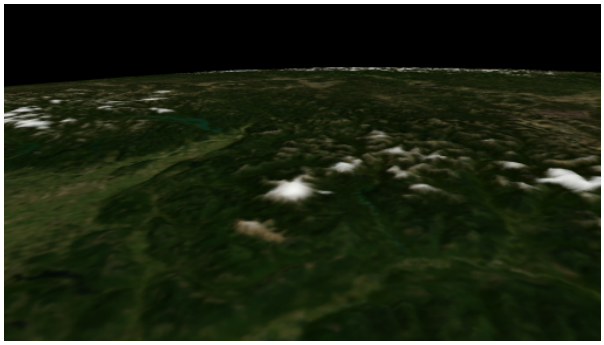


Figure 3.14: The diffuse color accumulation buffer

lated. All of these may be done without regard to the underlying geometric complexity of the scene. Because all composition is performed in a homogeneous texture space at the native resolution of the output device, no discontinuity will appear.

3.4.4 Output

Given accumulations of all relevant surface quantities mapped onto terrain geometry in screen space, the production of the final on-screen image is trivial. Figure 3.15 shows the composition of all data processed thus far. Geometry is colored and illuminated using the diffuse and normal buffers, and atmospheric scattering is applied. Other effects may be included here, such as high-dynamic-range tone mapping, spherical correction for dome display, and image interleaving for autostereo display.

Figure 3.16 reviews each of the steps described in this section. All six images show the same field of view, in the order of pipeline execution.

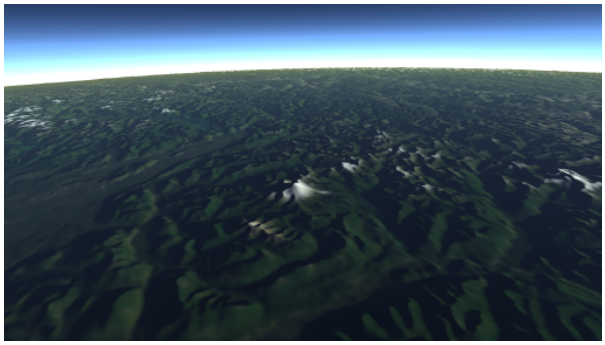


Figure 3.15: All buffers composed and illuminated with atmosphere.

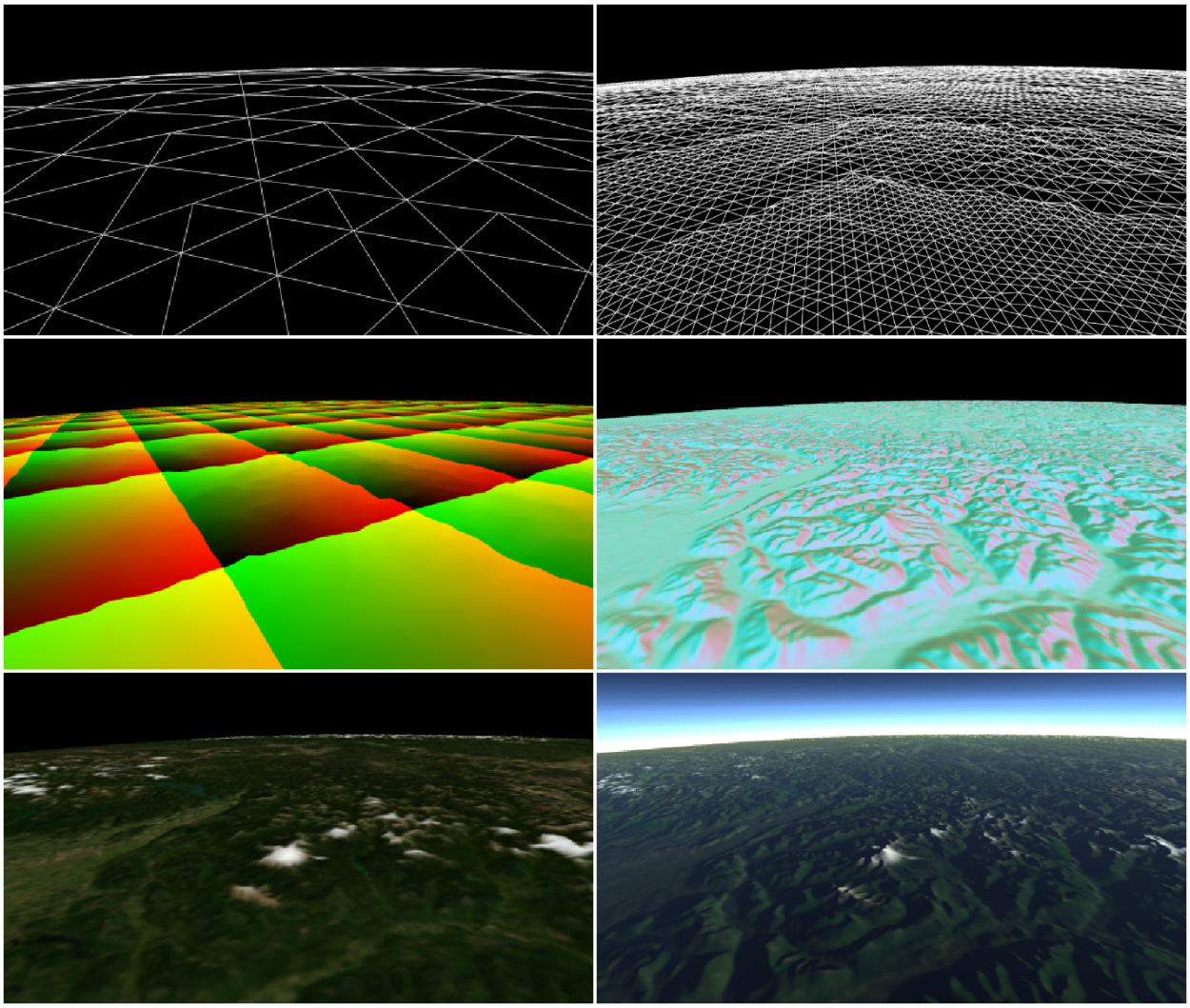


Figure 3.16: A review of the composition process. From the top-left: the CPU-generated patches, the GPU-displaced tessellation, the deferred texture buffer, the normal accumulation buffer, the diffuse accumulation buffer, and the final image.

Chapter 4

Composition Operations

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 techniques follow immediately. The following sections describe some of these. In all cases, the displayed figures were rendered by the implementation of this approach.

In an effort to make the display of the effects this method concrete and clear, this chapter employs the false-data example planet shown in Figure 4.1. All of these figures in this chapter were rendered using the 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 [33], 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 filtering so that non-uniform data sampling is made apparent in the shape of the rendered texels.

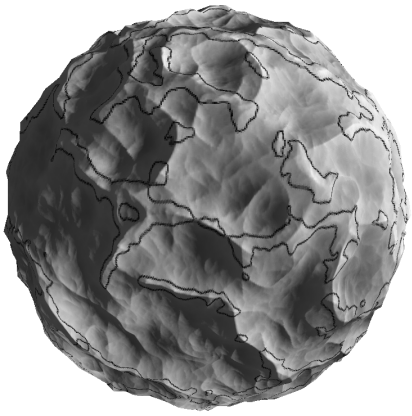


Figure 4.1: The 3D simplex noise planetary height map with normal and diffuse color maps, used for all subsequent figures.

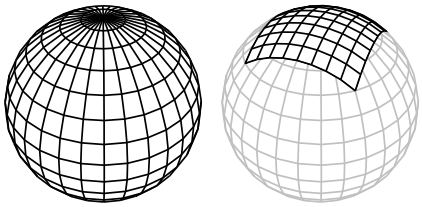
4.1 Projection Quality Adaptation

4.1.1 Background

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 Figure 4.2a. The Shuttle Radar Topography Mission (SRTM) [8] data set, a 1-arc-second 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° above and below the equator, and the spherical projection is optimal.

However, consider Blue Marble Next Generation (BMNG) [27], the familiar mosaic of color Earth imagery. It too is spherically projected, but extends all the way to the poles. Let us review Section 2.8.1: 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.



(a) Spherical projection (b) Polar projection

Figure 4.2: Common planetary projection types.

The correct solution for polar rendering is to use polar-projected source data, as in Figure 4.2b, where the sampling of the source most closely matches the sampling of the rendered image. The Landsat Image Mosaic of Antarctica (LIMA) [42] is an example of a high-resolution data set presented with polar projection. In particular, geoscientists at the Antarctic Geospatial Information Center (AGIC) at the University of Minnesota have 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) [47] data set is one example. It provides spherical projection of Mars height data up to 88° from the equator, filling the gap at each pole using data with polar projection.

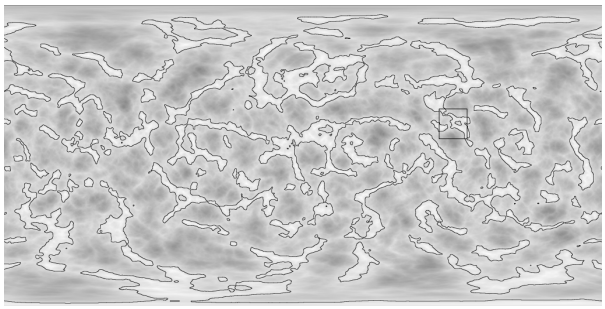
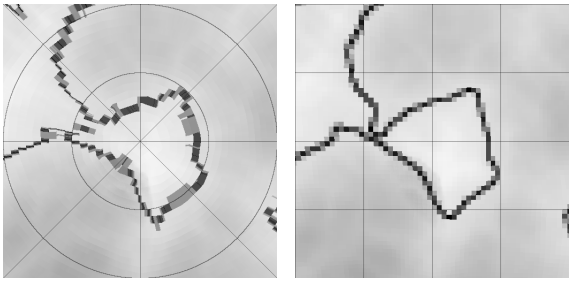


Figure 4.3: The spherical projection of the diffuse color of the example planet. The marked region corresponds to Figure 4.6.

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 the example planet. Figure 4.3 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 this method extend this discussion to terrain geometry as well as any other surface mapped quantities, including the normal maps used to produce these figures.

Figure 4.4a shows the south pole. The small contoured region there is stretched across the entire bottom of Figure 4.3. We see the extremely non-uniform 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.

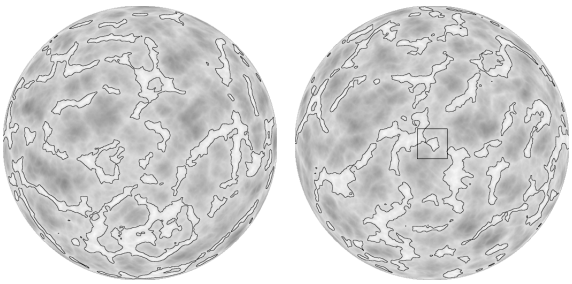
So, let us introduce the polar projection of the surface color map, as shown in Figure 4.5. Figure 4.4b shows this image mapped onto the sphere. Contrast



(a) Spherical (b) Polar

Figure 4.4: The south pole of the example planet, showing spherical (a) and polar (b) data mapped onto the sphere, contrasting the data sampling uniformity of each.

the uniformly-shaped pixels of Figure 4.4b with the stretched pixels of Figure 4.4a.



(a) North (b) South

Figure 4.5: Polar projection of the diffuse color of the example planet. The marked region corresponds to Figure 4.4.

While polar data map cleanly at the pole, sampling suffers elsewhere. Contrast the uniformity of the spherical data near the equator, shown in Figure 4.6a, with the non-uniform polar data at the same location in Figure 4.6b.

(a) Spherical (b) Polar

Figure 4.6: A region near the equator, showing spherical (a) and polar (b) data mapped onto the sphere, contrasting the data sampling uniformity of each.

4.1.2 Implementation

To produce a uniform sampling across the entire planet one must blend the spherical, north polar, and south polar data sets. This blending follows immediately from the accumulation mechanism described in Section 3.4.3. The only open question is choosing the weights of that blend. There are alternative approaches here.

The figures in this document use an extremely straightforward approach: cubic interpolation over distance. Figure 4.7 shows each of the three weighted terms separately, with their composition shown the right. In general, the weights need not add up to one. While (x, y, z) vector 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.

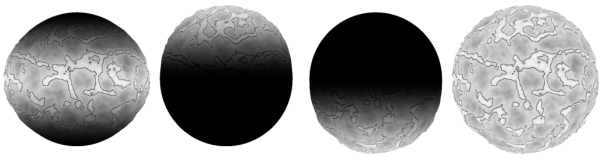


Figure 4.7: The cubic-weighted contributions of spherical, north polar, and south polar projected height and color data to the final planet.

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 3.4.2). 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.

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

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 over-lap, 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 it 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 3.4.3) it produces effects such as that shown in Figure 4.7 automatically.

Applied during the geometry displacement phase (Section 3.3.6), it enables the adaptive composition of height values, giving high quality geometry planet-wide. However, a bit of extra work is required. The layout of the geometry image buffer is logical, rather than spatial, so neighboring vertices are not adjacent in the buffer, the GPU's automatic finite difference derivatives are not valid, and texture coordinate derivatives must be explicitly computed during geometry generation.

4.2 Data Overlay

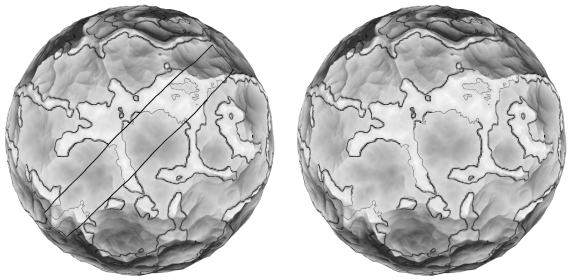
4.2.1 Background

Terrain visualization frequently requires the overlay of unregistered height and surface maps of differing resolution 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.

My 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, from launch in early 2009 through the next year. 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 StarRider 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 pre-existing lunar data, such as the Clementine and Lunar Orbiter data sets.

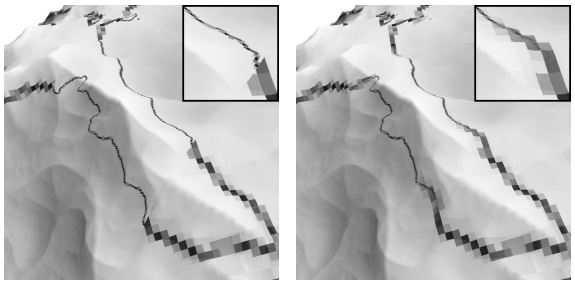
These examples follow straightforwardly from terrain composition. To show this, we may generate a projection of the sample planet similar to raw LROC output. A narrow strip runs across the planet, as though it followed the ground trace of a satellite in an inclined orbit. The color map appears in Figure 4.8. It is outlined in place in Figure 4.9a.

Figure 4.8: A strip of local high-resolution data, as collected by a satellite in an inclined orbit.



(a) With boundary indicated (b) Composed normally

Figure 4.9: A strip of local high-resolution data composed with global low-resolution data.



(a) Unblended (b) Blended

Figure 4.10: A close-up view of the border of the high-resolution strip of local data. Pixel size indicates sampling and resolution. The insets magnify the area of interest.

4.2.2 Implementation

Height and surface maps are accumulated 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. Figure 4.10a 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 Figure 4.10b. 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. 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.

4.3 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. Caching mechanisms handle data in *pages*, where a page is an atomic accessible subset of the data. A 2D data set such as a height or surface map is divided along a uniform grid, giving square pages. The reassembly of these data pages into a uniform on-screen whole follows immediately from the height map and surface map composition capability of the terrain composition algorithm.

4.3.1 Background

Data pages have a number of properties. The most significant characteristic of a page is its size. A given data set may divided into a large number of small pages, or a small number of large pages. Page size selection reveals the same type of trade-off as geometry batching (Section 2.4.1), and as such it is a choice necessarily guided by performance testing.

Small pages (approaching a single pixel in the limit) allow the application to access precisely the data that it requires, with low latency. However, if

these pages are too small then the run-time overhead of page acquisition becomes a bottleneck.

Large pages (approaching the size of the full data set in the limit) utilize bandwidth efficiently, but if they are too large then latency is high and both bandwidth and local storage go to waste, undermining the effectiveness of the cache hierarchy.

Because data is displayed at a variety of scales, a mipmapped approach (Section 2.4.2) to paging is necessary. If the planet is seen from afar then only low-resolution sub-sampled pages need be accessed. Just as with mipmapping, each level of the page "pyramid" gives four times the resolution (two-by-two) of the previous, with four times the total size, and thus four times the number of pages.

Finally, pages must have information about their neighbors. If the available data resolution does not meet the application's required resolution, then pixel values are interpolated by linear magnification filtering. If interpolation beyond the edge of a page should be required, then pixels of the adjacent page are needed. Rather than mandate that page be loaded, the edge pixels of all neighboring pages are appended. So a 1000×1000 pixel page has a true size of 1002×1002, as it includes one line of pixels from each of the four pages adjacent to it.

Large NASA and USGS data such as SRTM [8] and BMNG [27] are distributed in page form. SRTM pages are 1° square, inclusive, giving 3601×3601 pages for 1-arc-second US data and 1201×1201 for 3-arc-second International data. BMNG pages are 21600×21600. Testing with current hardware has shown an optimal page size around 500×500. OpenGL texturing prefers a power-of-two size, and this gives a useful page size of 512×512. Clearly, both SRTM and BMNG pages are far too large for real-time use, and they must be merged and re-sliced. This process is straightforward, and can be performed by a variety of off-the-shelf tools, including Global Mapper [14].

4.3.2 Implementation

The implementation of the data paging mechanism uses a variant on the technique developed by Lefohn et al [22] 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 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 8192 pixels square, and recent hardware supports 4096 pixels. In the 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 index texture behaves as a normal, unfiltered 2D mipmap. Rather than giving color (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 look-up process is performed by a GPU fragment shader, which may implement any mipmap access policy. The figures and performance measures shown in this document 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 [41], which depicts U.S. territories all over the world at a resolution of one arcsecond, 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 Figure 3.3. The solid angle subtended by this shell at the current view point, 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 Figure 3.8.

Figure 4.11 shows a view of the southwest United States with NED overlaid atop BMNG and SRTM, a total of 115GB of color, normal, and height data. In Figure 4.12 we see the 16 × 8-page RGB color cache used to render the scene. Note that both diffuse color and normal maps are 8-bit 3-component images, and they share this cache. Figure 4.13 shows the same view, with each page colored by its depth in the mipmap hierarchy. The smooth gradiation in color indicates the distance-appropriate transition in data resolution resulting from tri-linear mipmapping computed using screen-space derivatives. In the performance analysis of Chapter 5 we will take a detailed look at this mechanism in action using this same 115GB of tri-linearly-mipmapped real-world data.

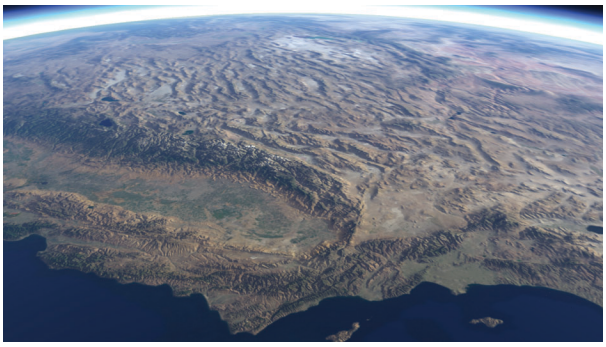


Figure 4.11: A view of the U.S. showing the NED, BMNG, and SRTM data sets, totaling 115GB.

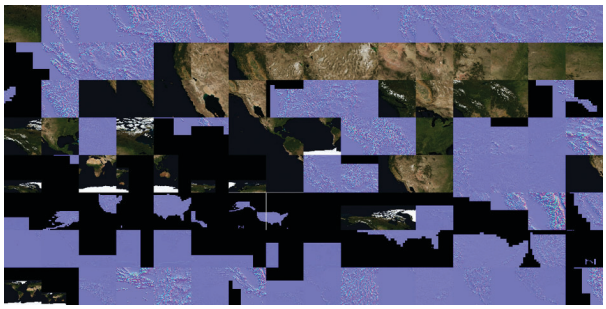


Figure 4.12: The 16 × 8 page cache used to render Figure 4.11 showing both diffuse color and normal map data pages.

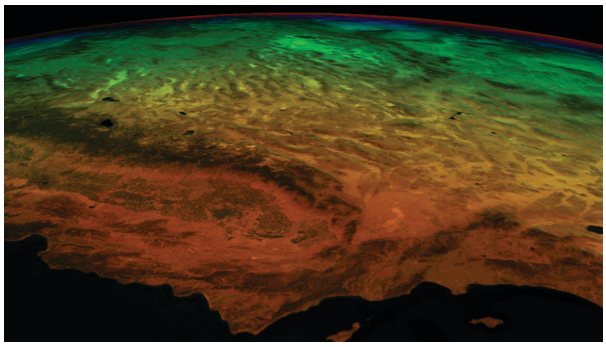


Figure 4.13: Figure 4.11 with each page of data colored by mipmap hierarchy depth, showing the smooth output of tri-linear mipmapping.

Chapter 5

Implementation and Results

5.1 Performance Measurement

In an effort to quantify the performance of the terrain composition algorithm and understand the variance in performance with different quality settings, let us define a benchmark. A scripted 4800-frame animation begins with a wide view of the Earth, moves in to a close-up view of Mount Rainier, and the moves back to the wide view along the same path. See Figure 5.1. 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 remains in the cache. This allows us to contrast the performance of the system under both I/O intensive and non-intensive circumstances. All frame time measurements are averaged over 10 frames, giving 480 data points per run.

The test configuration involves multiple gigapixel-scale data sets, paged and cached as described in Section 4.3, using a data overlay composition as described in Section 4.2. The base layer is the Shuttle Radar Topography Mission (SRTM) [8] 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) [41] 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 Blue Marble Next Generation (BMNG) [27] data set provides 24-bit RGB color covering the Earth at 30 arcseconds, for another 3.5 gigapixels.

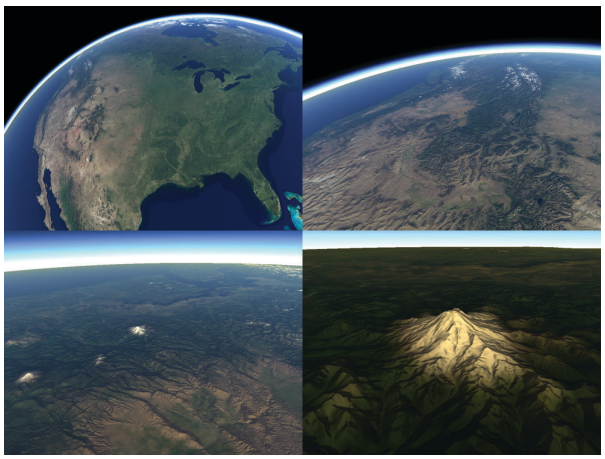


Figure 5.1: Frames 100 (a), 1600 (b), 2200 (c), and 2400 (d) of the 4800-frame benchmark animation.

In total, this is 115GB of raw data. Each of these data sets was prepared using Global Mapper. The raw data were received from AGIC in the form of Global Mapper project files, which were loaded and exported as tiles. Some difficulty was encountered when Microsoft Windows' filesystem limitations were exceeded. The maximum number of files allowed in a single directory was insufficient. To resolve this, Global Mapper was used to export 5100×5100 tiles, which were then further subdivided to 510×510 by a custom Linux command-line tool. Tile boarders were then added to these, and the mipmap hierarchy was generated by another custom command-line

tool. In the event that the number of tiles does not exceed Windows' limits, Global Mapper would generate the borders and mipmaps itself. This process produced 189,000 mipmapped tiles, each 512 pixels square. These tiles were compressed to the PNG form, consuming 38GB of disk space.

The input data types allowed by this process include 8, 16, and 32-bit signed and unsigned integers, as well as floating point values. Images may have between one and four channels. Any image data type loaded and stored by Global Mapper is straightforwardly adaptable to this implementation. Global Mapper does allow a variety of vector data types which the terrain composition algorithm does not consider, though the implementation trivially allows such data to be rendered with its output.

5.1.1 Baseline performance

The primary test hardware is a dual AMD Opteron 250 at 2.4GHz with 4GB of RAM and an NVIDIA GeForce 8800 GTX. The baseline run of this configuration has a resolution of 1024×768, a refinement depth $d = 4$, a patch seed count $n_p = 256$. 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 Figure 5.2.

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

Note that all graphs presented here use the same vertical scaling, where the top of the graph represents a 30Hz refresh rate, and the middle of the graph represents a 60Hz refresh rate. Figure 5.2 shows performance comfortably better than 60Hz throughout the run.

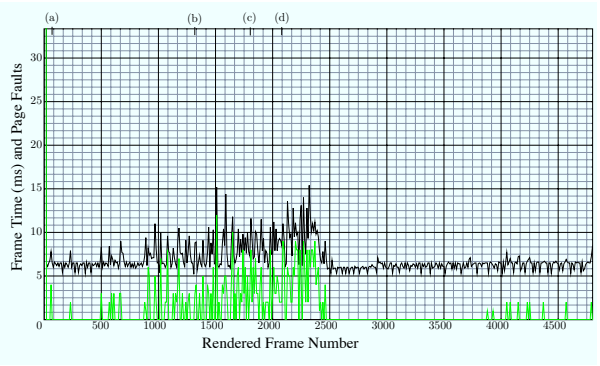


Figure 5.2: Baseline performance at 1024×768 ($d = 4$, $n_p = 256$) measured over time. Frame time (ms) is shown in black and page fault count in green. Marked frames (a), (b), (c), and (d) refer to Figure 5.1. Note the generally level trend, with disturbances correlating frame time with page faults.

5.1.2 Variance with resolution

Now let us vary the resolution of the display without varying the complexity of the geometry. The results are shown in Figure 5.3. 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 3.2) and geometry generation phase (Section 3.3). We see performance level at 5ms. This is the major fraction of the 6ms performance at 1024×768 (again in black), but it is fortunately a constant dependant only upon geometric complexity.

The primary impact of increasing the display resolution to 1920×1080 (Figure 5.3 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.

We can still see a consistent minimum frame time of around 10ms in this circumstance, due to fragment processing overhead. Given the assumption of 5ms of geometry overhead inferred from the 320×240 results, we see the increasing fragment cost match the geom-

etry cost at this resolution. As resolution increases from here, the balance tends toward fragment processing.

The important case of stereoscopic rendering sees a benefit here. A stereo display entails the rendering of the scene once for each of the user's two eyes. This effectively doubles the resolution of the display, and we would see the frame time roughly double with this added fragment load. However, because of the proximity of the user's eyes, and the resulting overlap between the two views, there is a great deal of data coherence between the two renderings. The data demand for two eyes would match that for a single eye, and the increase in resolution due to stereoscopic rendering would not incur the same I/O penalty as an equivalent doubling in monoscopic resolution.

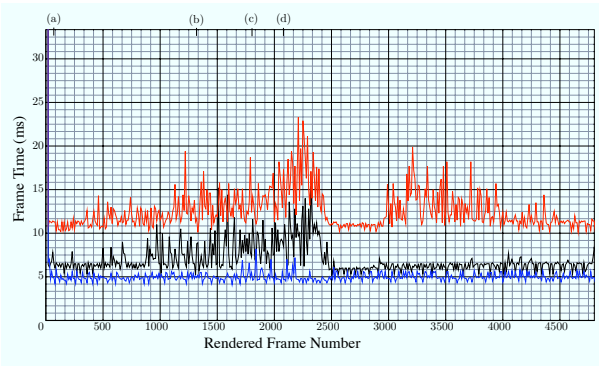


Figure 5.3: Frame time (ms) measured over time at 1024×768 in black, at 320×240 in blue, and 1920×1080 in red. Frame time varies with pixel count, and increased resolution incurs greater data demand.

5.1.3 Variance with geometry

Now let us vary the parameters that determine geometric complexity while holding the display resolution constant at 1024×768. First, double the number of seed patches from $n_p = 256$ to $n_p = 512$ while holding the refinement depth d constant. This has the effect of doubling the number of rendered vertices from $39,168$ to $78,336$ (Section 3.2). We see this doubling borne out in Figure 5.4 with the frame time graph translated upward.

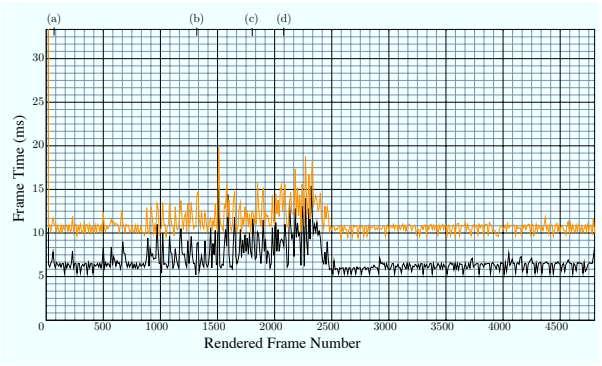


Figure 5.4: Frame time (ms) measured over time at 1024×768 with seed patch count $n_p = 256$ in black and $n_p = 512$ in orange. Doubling the geometry doubles the frame time.

Orthogonally to this, increase the refinement depth from $d = 4$ to $d = 5$ while holding the seed patch count n_p constant. This has the effect of increasing the number of vertices 3.6 times, from $39,168$ to $143,616$ (Section 3.3). Again, we see the impact of this clearly in Figure 5.5. If we infer from Figure 5.3 that 1ms of the frame time may be attributed to the overhead of rendering 1024×768 pixels, then the remaining 17ms of $d = 5$ frame time is quite close to 3.6 times the 5ms 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, when doubling n_p, 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. CPU monitoring shows the load of the rendering thread to be negligible. However, one can imagine a circumstance where careful manipulation of n_p and d may tune the CPU-GPU balance and improve throughput.

5.1.4 Variance with data

The data caching mechanism ensures that the total size of a given data set does not impact general per-

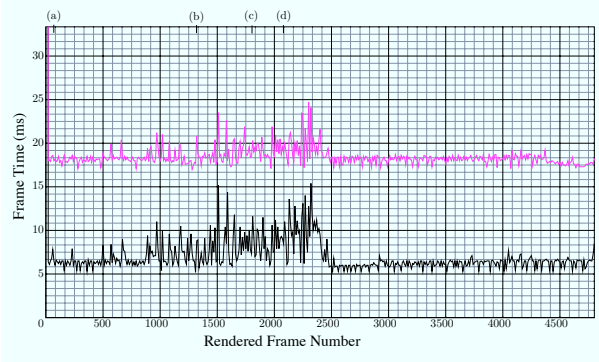


Figure 5.5: Frame time (ms) measured over time at 1024×768 with patch subdivision depth $d = 4$ in black and $d = 5$ in pink. The $3.6\times$ increase in geometry is seen in the frame time.

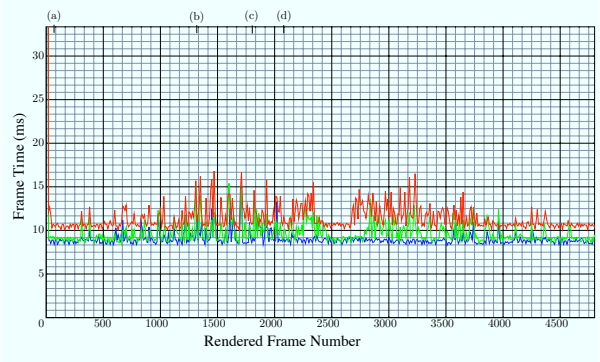


Figure 5.6: Frame time (ms) measured over time at 1920×1080 with SRTM only in blue, SRTM+NED height in green, and SRTM+NED normal in red. Additional data incurs only minor penalty.

formance. However, data layering does incur overdraw, so performance does vary with the total number of data sets composed. To quantify this, we run the 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.

Figure 5.3 showed that fragment processing is not a serious bottleneck at 1024×768, so to see a distinction, we perform this test at 1920×1080. Figure 5.6 shows this result in blue. As expected, 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, apply the NED height map atop the SRTM without the NED normal map. This is shown in Figure 5.6 in green. 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, apply the NED normal map atop the SRTM without the NED height map. This is shown in red. Surface map overdraw incurs the rendering of another full-screen rectangle, which results in an overall increase in frame time of 2ms.

5.1.5 Performance Qualities

These results reveal a balanced pipeline at the 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 [5] tend toward a geometric bottleneck, and well-batched GPU-based algorithms such as geomipmapping [4] lead to a pixel bottleneck. By offloading geometry processing to the GPU, this approach provides 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 generated geometry buffers and deferred shading buffers, may require synchronization.

It would be careless to overlook these issues, so Figure 5.7 displays the results of early testing of the algorithm in a multi-GPU environment. The base-

line GeForce 8800 GTX is 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.

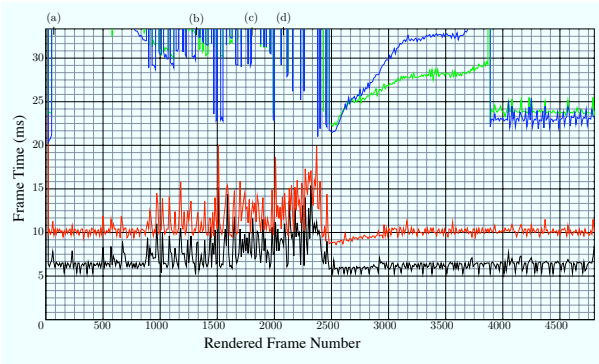


Figure 5.7: Multi-GPU performance comparison: single-GPU shown in black, multi-GPU in single mode in red, alternate-frame in green, split-frame in blue. The pathological response warrants further investigation.

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 the algorithm introduces no inter-frame 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 intra-frame data dependency, one 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 23ms 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 2500. Its slow crawl from 23ms up to and beyond 30ms during this period remains unexplained.

In the context of standard forward rendering, the current hardware industry trend toward multi-GPU configurations is clearly beneficial. However, 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 the implementation to treat multiple GPUs as wholly separate renderers, each with an independant framebuffer.

5.2 Displays and installations

The parallel-rendering implementation of the scalable terrain composer has been adapted to a variety of displays and has seen use in a number of demonstrations. Each of these uses the BMNG [27] surface map atop the SRTM [8] height map, both at a resolution of 86400×43200.

Figure 5.8: The 60-panel Varrier^{TM} at Calit2

The first installation, shown in Figure 5.8 used the 60-panel Varrier^{TM} autostereoscopic virtual reality display [35] at Calit2 on the campus of UCSD.

This display is driven by 15 rendering nodes, each with a pair of NVIDIA GeForce 7900GTXs driving

four 1600 × 1200 panels, giving a total resolution of 50 megapixels per eye.

This is a challenging system, as each render node must handle eight distinct view frusta and 7 megapixels of sub-pixel autostereo interleaving [20].

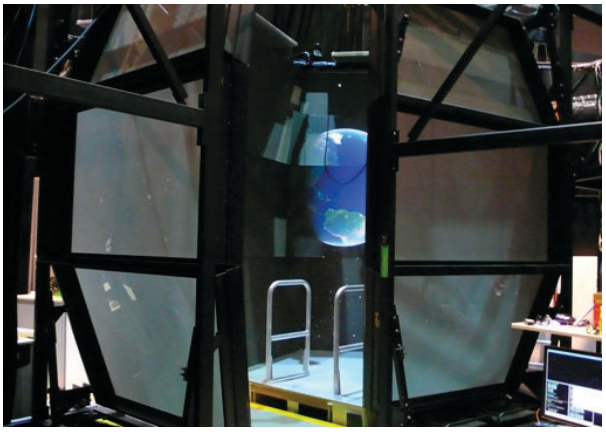


Figure 5.9: The StarCAVE at Calit2

Also at Calit2 is the StarCAVE, a 15-screen cylindrical polarized passive stereo VR display, shown in Figure 5.9. Driven by 15 render nodes, each with an NVIDIA Quadro 5600 driving two projectors at 1920 × 1080, this system has high pixel throughput, and runs very well.

Figure 5.10: The Transporter prototype in the Space Visualization Lab at the Adler Planetarium

The Space Visualization Lab at Chicago's Adler Planetarium has also supported the development and use of this software. The Mars Transporter is on exhibit at the Adler Astronomy Museum, and as the usability of the terrain composer improves it will act as a drop-in replacement, significantly expanding the quality, scope, and flexibility of that experience. Until that time, the implementation is on display on the SVL's prototype Transporter, shown in Figure 5.10.

Figure 5.11: The DeFiniti theater at the Adler Planetarium

A much more ambitious demonstration has also been developmented at the Adler. SVL staff have installed a small cluster of 3 render nodes driving the six inputs to the DeFiniti theater, a 55-foot digital dome seating around 200. Each node has a pair of NVIDIA GeForce 8800GTXs each driving one 1600 × 1200 projector.

The challenge lies in pre-distorting the rendered image to map correctly onto the spherical surface of the display, and blending adjacent images to obscure the seams between projectors. A GPU-based solution to this problem has been developed and tested in place, as shown in Figure 5.11. The performance penalty of spherical correction and blending is found to be negligible.

Calit2's OptIPortable display has taken this software on the road. 12 render nodes, each with an NVIDIA GeForce 8600GT drive 15 panels at 1920 × 1080. This was demonstrated in the SDSC/Calit2/EVL booth at Supercomputing '07, as shown in Figure 5.12. It was also demonstrated in

Figure 5.12: The OptIPortable at Supercomputing '07

the NSF booth at the 2008 meeting of the American Academy for the Advancement of Science, Figure 5.13.

Figure 5.13: The OptIPortable at AAAS '08

Chapter 6

Conclusions and Future Work

This work has demonstrated a mechanism for the real-time manipulation and display of very large scale 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.

This discussion has detailed a number of areas where future work is required. In particular, the analysis of error and the effects of the non-uniform sampling of height data are critical to the acceptence of the algorithm as a reliable tool. In addition, while the generality of terrain composition provides a path to an elegant implementation of geomorphing, this has yet to implemented. Finally, the adaptation of this 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.

In particular, the effects of time-varying data will be tested. The ability to apply alpha blending to height and surface data provides a means of interpolating between data sets representing different points in time, both in geometry and in imagery. In this way, a smoothly-animated approximation of time-varying data may be presented without popping from step to step. The twelve monthly versions of Blue Marble Next Generation are a prime example of a data set that would benefit from this. The challenge arrises due to the doubling of data needed to represent both the beginning and the end of the interpolation.

Also, layered data will be explored. This too is primarily a data scalability issue, but with display and user interface issues involved. A straightforward presentation of layered data would have upper layers occlude lower layers, so informative and efficient mechanisms for presenting multiple simultaneous layers must be implemented. Meanwhile, an affordance by which the user selects and peels away layered data must be devised. Examples of such layered data sets include the ice surface and land of Antarctica, and the bathymetry beneath the surface of the oceans.

Our partnerships with the Antarctic Geospatial Information Center (AGIC) and the Adler Planetarium and Astronomy Museum 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.

Bibliography

[1] Arul Asirvatham and Hughues Hoppe. *Terrain Rendering Using GPU-Based Geometry Clipmaps*, volume 2 of *GPU Gems*, chapter 2, pages 27–45. Addison-Wesley, 2005.

[2] James F. Blinn. Simulation of Wrinkled Surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292, New York, NY, USA, 1978. ACM.

[3] Malte Clasen and Hans-Christian Hege. Terrain Rendering using Spherical Clipmaps. In *Proceedings of Eurographics/IEEE-VGTC Symposium on Visualization*, 2006.

[4] Willem H. de Boer. Fast Terrain Rendering Using Geometrical MipMapping. flipcode.com, October 2000.

[5] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. Technical Report UCRL-JC-127870, Lawrence Livermore National Laboratory, October 1997.

[6] George Eckel and Ken Jones. OpenGL Performer Programmer's Guide. Technical Report 007-1680-060, Silicon Graphics Inc., 1997.

[7] Cass Everitt and Mark J. Kilgard. Practical and Robust Stenciled Shadow Volumes for Hardware-accelerated Rendering. *Arxiv preprint cs.GR/0301002*, 2003.

[8] T.G. Farr and M. Kobrick. The Shuttle Radar Topography Mission. *Reviews of Geophysics*, 45(2), 2005.

[9] R.L. Ferguson, R. Economy, W.A. Kelly, and P.P. Ramos. Continuous Terrain Level of Detail for Visual Simulation. *Proceedings, IMAGE V Conference*, pages 144–151, 1990.

[10] R. Buckminster Fuller. Cartography, January 1946. US Patent #2,393,676.

[11] Eric Gaba. Fuller Projection, September 2006. Creative Commons Attribution and Share Alike license.

[12] Michael Garland and Paul S. Heckbert. Fast Polygonal Approximation of Terrains and Height Fields. Technical Report CMP CS 95 181, Carnegie Mellon University, September 1995.

[13] Michael Garland and Paul S. Heckbert. Survey of Surface Approximation Algorithms. Technical Report CMP CS 95 194, Carnegie Mellon University, May 1997.

[14] Global Mapper Software. Global Mapper. globalmapper.com.

[15] Google. Google Earth. earth.google.com.

[16] Lok M. Hwa, Mark A. Duchaineau, and Kenneth I. Joy. Adaptive 4-8 Texture Hierarchies. *Vis*, 00:219–226, 2004.

[17] IEEE Task P754. *ANSI/IEEE 754: Standard for Binary Floating-Point Arithmetic*. IEEE, New York, August 1985.

[18] John Kessenich. The OpenGL Shading Language version 1.20. opengl.org, September 2006.

[19] Mark J. Kilgard, Ralf Biermann, and Derek Cornish. ARB pixel buffer object. opengl.org, December 2004.

[20] Robert Kooima, Tom Peterka, Javier Girado, Jinghua Ge, Dan Sandin, and Tom DeFanti. A GPU Sub-pixel Algorithm for Autostereoscopic Virtual Reality. *Virtual Reality Conference, 2007. VR'07. IEEE*, pages 131–137, 2007.

[21] Stanford University Graphics Lab. BrookGPU. graphics.stanford.edu.

[22] Aaron E. Lefohn, Shubhabrata Sengupta, and John D. Owens. Resolution-matched Shadow Maps. *ACM Trans. Graph.*, 26(4):20, 2007.

[23] P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hodges, N. Faust, and G.A. Turner. *Real-time, continuous level of detail rendering of height fields.* ACM Press New York, NY, USA, 1996.

[24] Frank Losasso and Hugues Hoppe. Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 769–776, New York, NY, USA, 2004. ACM.

[25] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM.

[26] Microsoft. DirectX. microsoft.com.

[27] NASA. Blue Marble Next Generation. nasa.gov, 2005.

[28] NVIDIA. GeForce3: The Infinite Effects GPU. nvidia.com, February 2001.

[29] NVIDIA. GeForce 8800 Ultra. nvidia.com, November 2006.

[30] NVIDIA. Compute Unified Device Architecture. nvidia.com, February 2007.

[31] Sean O'Neil. A Real-Time Procedural Universe, Part Two: Rendering Planetary Bodies. gamasutra.com, August 2001.

[32] Sean O'Neil. *Accurate Atmospheric Scattering*, volume 2 of *GPU Gems*, chapter 16. Addison-Wesley, 2005.

[33] Kenneth Perlin. Standard for Perlin Noise, March 2005. US Patent #6,867,776.

[34] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-D shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, 1990.

[35] Dan Sandin, Todd Margolis, Jinghua Ge, Javier Girado, Tom Peterka, and Tom DeFanti. The Varrier^{TM} Autostereoscopic Virtual Reality Display. *Proceedings of ACM SIGGRAPH 2005*, 24(3):894–903, 2005.

[36] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification. opengl.org.

[37] Irwin Sobel. An Isotropic 3×3 Image Gradient Operator. *Machine Vision for Three-Dimensional Scenes*, pages 376–379, 1990.

[38] Alexander S. Szalay, Jim Gray, George Fekete, Peter Z. Kunszt, Peter Kukol, and Ani Thakar. The Hierarchical Triangular Mesh. *Mining the Sky: Proc. of the MPA/ESO/MPE workshop*, pages 631–637, August 2005.

[39] Thatcher Ulrich. Rendering Massive Terrains using Chunked Level of Detail Control. In *SIGGRAPH 2002 Course Notes*, July 2002.

[40] NASA/Arizona State University. Lunar Reconnaissance Orbiter Camera. lroc.sese.asu.edu.

[41] U.S. Geological Survey (USGS). National Elevation Database. ned.usgs.gov, 1999.

[42] U.S. Geological Survey (USGS). Landsat Image Mosaic of Antarctica. lima.usgs.gov, 2007.

[43] Lance Williams. Casting Curved Shadows on Curved Surfaces. *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274, 1978.

[44] Lance Williams. Pyramidal Parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, 1983.

[45] Matthias Wloka. Batch, Batch, Batch: What Does It Really Mean? In *Game Developer's Conference*, 2003.

[46] Mason Woo, Jackie Neider, and Tom Davis. *The OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley, 1997.

[47] M. T. Zuber, D. E. Smith, S. C. Solomon, D. O. Muhleman, J. W. Head, J. B. Garvin, J. B. Abshire, and J. L. Bufton. The Mars Observer Laser Altimeter Investigation. *Journal of Geophysical Research*, 97:7781–7797, May 1992.