University of Nevada
Reno

Projective Grid Mapping for Planetary Terrain

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

by

Joseph D. Mahsman

Dr. Frederick C. Harris, Jr., Thesis Advisor

December 2010

University of Nevada, Reno
Statewide • Worldwide

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

JOSEPH DAVID MAHSMAN

entitled

Projective Grid Mapping For Planetary Terrain

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Frederick C. Harris, Jr., Ph. D., Advisor

Daniel Coming, Ph. D., Committee Member

Sergiu Dascalu, Ph. D., Committee Member

Scott Bassett, Ph. D., Graduate School Representative

Marsha H. Read, Ph. D., Associate Dean, Graduate School

December, 2010

Abstract

Planetary terrain presents a number of issues that terrain rendering techniques commonly fail to address: displaying a vast amount of data at both small and large scales, using datasets in an appropriate map projection for the geographic area depicted, and rendering terrain features that rise from behind the horizon, such as tall mountains. The utility of planetary terrain rendering increases with both the availability of planetary data and the ubiquity of powerful, consumer-level GPUs. In addition, the visualization of full-scale planetary bodies has benefits for education and science. Projective Grid Mapping (PGM) is a GPU terrain rendering technique that combines the advantages of ray casting and rasterization, providing gradual level-of-detail transition and steady framerates. We formulate a version of the projective grid mapping algorithm for planetary terrain; this involves the view-dependent creation of reference spheres for ray casting and the construction of a sampling camera that approximately samples the geographic area of the planet that could possibly affect the final image. In addition, we reduce ray-sphere intersection to two dimensions, which simplifies implementation on the GPU. In order to create a complete planetary terrain renderer, we combine this spherical PGM technique with deferred texturing, which composites overlapping datasets. The visualization operates on the desktop and in two virtual reality displays. We achieve efficient framerates for various grid and screen resolutions, and we demonstrate high-quality views of Mars for both the equatorial and polar regions.

Acknowledgments

This work is funded by NASA EPSCoR, grant # NSHE 08-51, and Nevada NASA EPSCoR, grants # NSHE 08-52, NSHE 09-41, NSHE 10-69.

Thanks to my committee: Dr. Frederick C. Harris, Jr., Dr. Scott Bassett, Dr. Daniel Coming, and Dr. Sergiu M. Dascalu. Thanks to my advisor, Dr. Frederick C. Harris, Jr., for his patience in allowing the ideas within this thesis to evolve.

Cody White and Roger Hoang deserve many thanks for giving hours of discussion and feedback and for contributing code to this project.

Thanks to Michael Leverington for showing me how problems are actually solved.

Thanks to all my friends, co-workers, and everyone from the High Performance Computing and Visualization lab (HPCVIS) at the University of Nevada, Reno and the Center for Advanced Computation Visualization and Modeling (CAVCaM) at the Desert Research Instute for their encouragment throughout the various stages of writing a thesis. Also, thanks to everyone who read this document and provided feedback.

Finally, thanks to my family for supporting me throughout many years of college, and especially to my mother for never letting me go home empty handed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past three decades an imposing volume of terrain rendering research has taken place. Despite this level of attention, only a portion of the research addresses the obstacles of full-scale planetary rendering.

A critical aspect of modeling a planetary body is the use of numerous height and color datasets. When two datasets share geographic regions, their data must be combined. In addition, full resolution datasets are too large to fit in both core memory and graphics memory. Therefore, data must be streamed from disk to core memory to graphics memory as needed.

Planetary rendering requires a scale model that allows precision within one meter. Therefore, the geographic location and extent of the datasets must be taken into account. Also, a common problem in planetary visualization occurs when equirectangular projected data, most accurate at the equator, is used for areas near the poles. A suitable map projection must be used for the area depicted.

The scale requirement and the limitations of 32-bit floating point numbers present precision issues at meter resolutions, specifically when the radius of the planet is on the order of millions of meters. For example, the equatorial radius for Mars is about 3,396,000 meters [58], a number that uses the seven digits of precision afforded by the IEEE 32-bit floating point data type [33]. Although recent GPUs support 64-bit floating point values, they do so at reduced speed. Also, depth buffer precision and rendering near and far objects along with the terrain are issues for scale models. In particular, the triangles of the mesh should not exhibit z-fighting, and the terrain should have the ability to occlude other objects in the simulation.

Finally, most algorithms focus on the planar case in which vertices are displaced vertically from a plane using a height map. However, a planet is more accurately represented with a curved surface. In geographic mapping, a sphere is often used as a datum or reference surface for geographic positions and elevations [32]. In addition, ray-based terrain approaches [14, 17, 55] face the problem of capturing the terrain features that rise from behind the horizon of the sphere, such as a mountain or a cliff.

Planetary terrain algorithms must also meet the standards of other terrain algorithms with respect to continuous level of detail (CLOD). These algorithms commonly generate different levels of detail based on the viewer's position. Lindstrom *et al.* [36] describe the criteria for CLOD: the generated mesh is smooth both spatially and temporally, the transition between levels of detail is gradual, and the number of polygons differs very little between frames as the viewer moves.

Another requirement relates to the appropriate use of graphics processing capabilities. The steady improvement of graphics hardware has led to the communication pipe between the CPU and the GPU becoming the application bottleneck. The algorithms by Hu *et al.* [29] and Kooima *et al.* [34] are representative of the recent effort to operate entirely on the GPU to reduce the effect of these bottlenecks. By moving work to the GPU, the CPU is free for time-critical tasks that are unsuitable or impossible for the GPU, such as user interaction and I/O operations. The parallelism of GPUs also makes representations on the GPU more flexible for image processing operations, as required for data composition.

The method used to generate a terrain mesh affects the ability to modify the mesh at run time. Cignoni *et al.* [5] describe a CPU-based algorithm that creates a complex mesh hierarchy as a preprocessing step. The result is a mesh that is appropriate for the current viewpoint. However, using a mesh hierarchy encumbers data composition. Kooima *et al.* [34] describe a GPU-based algorithm that generates a mesh almost entirely using the GPU. The generated mesh is stored and used as a texture, and subsequent data composition operations generate the final image. These composition operations are implemented as a series of deferred texturing passes.

Projective Grid Mapping (PGM) [37, 38] is an algorithm that operates entirely on the GPU and attempts to combine the advantages of ray casting and rasterization. A screen-space grid is projected onto the terrain reference plane using ray casting, where each grid point represents a vertex of the mesh. The vertices are offset by a height map in graphics memory. The advantage of PGM is that it involves sparse CPU/GPU communication because it operates only on GPU memory. It follows that the GPU representation of the mesh allows for data composition. However, previous PGM algorithms for terrain do not solve the problem of choosing a reference surface for ray casting; this is typically a plane for most applications, but for planetary rendering it is a sphere. In a scale planetary model, the minimum and maximum heights can differ by tens of thousands of kilometers. The choice of reference sphere affects how far the triangles are displaced with respect to the viewer and therefore whether the resulting triangles are approximately pixel-sized in screen space.

We present a method of mesh generation using PGM that addresses most of the issues of planetary rendering just described. A scheme for streaming data between disk, core, and graphics memory is not included, however both Livny *et al.* [37] and Kooima *et al.* [34] implement GPU caching algorithms that are compatible with our approach. These streaming methods are compatible since they use a texture cache on the GPU and integration with our algorithm would only require specifying the portions of the texture cache to be used in the shader code. In addition, although progress was made in addressing floating point precision, more can be done with this algorithm to approach accurate visualization within a meter.

In our algorithm, the creation of the mesh occurs entirely on the GPU for each frame, and the mesh is treated as a number of textures that remain in graphics memory. The algorithm operates in approximate screen space for height composition and screen space for color and normal composition. Additionally, programmable vertex and fragment shaders are used, and these shaders can be implemented on hardware that supports vertex texture lookups.

This work includes several contributions:

- We adapt PGM for the sphere by using trigonometric functions to solve the two-dimensional case for ray-sphere intersection. Working in two dimensions simplifies shader logic and enables interpolation of ray attributes, a property important for adequate sampling described later (Chapter 3).

- As noted previously, a reference sphere that minimizes the displacement distance of mesh vertices should be used for PGM. We compare two methods for choosing such a reference sphere (Section 3.4.1).

- Ray casting the sphere fails to capture the parts of the terrain that rise from behind the horizon. We present a method for calculating an additional reference sphere that, when ray cast, samples the area beyond the horizon for a given reference sphere (Section 3.4.2).

- We generate a camera that approximately samples the portion of the planet that affects the final image. This method involves intersection of infinite frusta that share an apex (Section 3.5).

- We modify equations for two common map projections, equirectangular and polar stereographic, from their formal descriptions into versions suitable for programmable shaders (Section 4.1).

- In generating normal maps, we treat data in a polar stereographic projection differently due to its unique data layout in relation to geographic coordinates (Section 4.2).

- We describe a method for correctly occluding secondary objects with the terrain for both small and large distances from the viewpoint. The goal is to avoid z-fighting in both the terrain and the secondary objects, and render every surface in the scene at the correct depth (Section 4.3).

Our algorithm performs PGM and, similarly to Kooima *et al.* [34], applies a number of deferred texturing passes to the resulting buffers (Chapter 3). First, the algorithm determines two reference spheres and a sampling camera that lead to an adequate sampling of the terrain. PGM is then performed by ray casting each grid point in the projective mesh onto the two spheres and interpolating. After the grid is projected onto the sphere, a series of deferred texturing passes are executed that result in the final image.

Planetary terrain rendering involves various types of data processing (Chapter 4). Equations for map projection are implemented on the GPU. These equations require constants that are calculated for each dataset at initialization and uploaded to the GPU during rendering. Normal maps that use a polar stereographic projection must be reconciled with normals that use an equirectangular projection to correctly apply lighting. Finally, a mechanism should be provided so that secondary objects, rendered after the terrain, can be correctly occluded with the terrain.

We implemented the algorithm as a terrain rendering library in order to meet a number of requirements (Chapter 5), and we found that the algorithm leads to reasonable framerates for a number of standard screen resolutions (Chapter 6).

Chapter 2

Background

The defining feature of a terrain algorithm is how it generates a mesh that approximates the surface while minimizing the cost of rendering. An initial approach begins with a regular grid of vertices that define a mesh, as shown in Figure 2.1, with the input height and color maps defining the terrain surface.

A landform is described by height and color data, often stored as grids of data called rasters. A height map contains one raster where the value of each pixel represents a height offset from the reference plane. A color map comprises one raster for greyscale data and three rasters for full-color satellite imagery. In addition, each pixel in a raster corresponds to a geographic area described by a pair of geographic coordinates (λ, ϕ) which are, respectively, longitude and latitude.

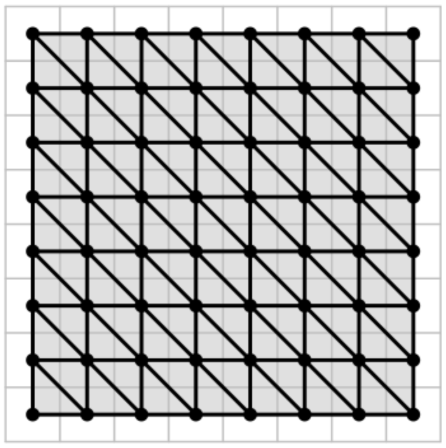


Figure 2.1: A regular grid of vertices form a triangle mesh. [33]

The mesh, originally a flat surface, is displaced according to the height map in a technique called displacement mapping [9, 10]. This is possible since each vertex corresponds to exactly one grid cell in the height map. Conceptually, the mesh begins as a plane, and each vertex is displaced to the correct height. Planetary rendering, however, must instead displace the mesh from a sphere.

Rendering the entire height map as a triangle mesh is inefficient. The entire mesh is sent to the graphics system every frame even though not all of the terrain may be visible. The visible triangles may appear so close and so large to the viewer that the discrete nature of the surface is apparent. Alternatively, the visible triangles may be so small that they affect less than a pixel of the final image, leading to an inefficient use of the graphics system's triangle throughput.

Advanced algorithms use various levels of detail (LODs) to reduce processing time while maintaining image quality. Areas far from the viewer may be rendered with lower detail using a few large triangles without a loss of image quality. Areas close to the viewer must be rendered using many small triangles to maintain image quality. The vertices of the generated mesh adequately sample the raster data, and each triangle equally contributes to the image. LOD algorithms are usually qualified as being "adaptive," meaning that new LODs are generated as the view changes.

Hardware capability motivates the design of mesh generation algorithms. As graphics hardware changes, the optimal strategy for rendering terrain changes. CPU-based approaches (Section 2.1) are designed to generate the fewest triangles that approximate the terrain to an error threshold. Batched approaches (Section 2.2) take advantage of on-board graphics memory and triangle throughput of the GPU. Approaches using programmable GPUs (Section 2.3) allow terrain representations to be generated almost entirely on the GPU. These methods allow for composition of multiple datasets and terrain deformation on the GPU. Composition is important for removing distortion, which is common to planetary terrain rendering. In addition, the CPU is free for time-critical tasks, such as streaming massive amounts of terrain data from disk. These features are important for planetary terrain renderers (Section 2.4).

2.1 CPU-based Approaches

Prior to 1999, the transformation and lighting stages of the graphics pipeline were implemented in software, limiting the triangle throughput of the graphics system. Terrain algorithms were designed to generate meshes with an optimal number of triangles while maintaining a user-specified precision with respect to the input height data.

The two formulations of these algorithms are mesh refinement and mesh decimation [27]. Mesh refinement begins with a low-resolution mesh and adds triangles until the specified maximum error threshold is satisfied. Mesh decimation works in the opposite direction, beginning with the highest resolution mesh and removing triangles. Figure 2.2 demonstrates refinement from left to right and decimation from right to left. Refinement and decimation are inverse operations as long as the same vertices are added or removed from the mesh. The mesh resulting from these operations is called a triangulation, named for the basic primitive that the graphics system uses.

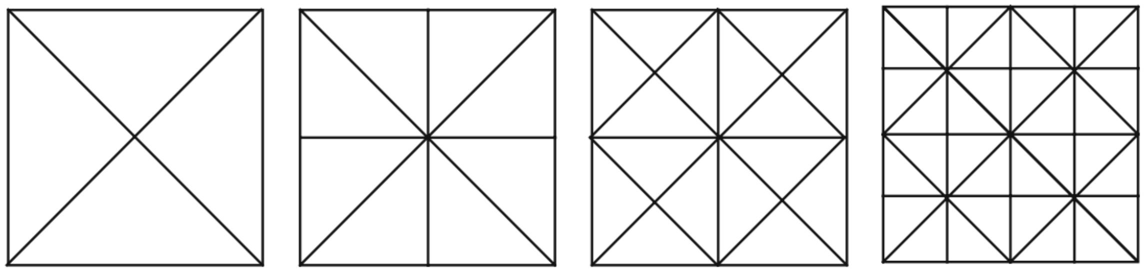


Figure 2.2: A set of successively refined meshes. [53]

2.1.1 Irregular Meshes

An irregular mesh, also known as a triangulated irregular network (TIN), is an unconstrained surface representation. Algorithms involving TINs place few constraints on the size and distribution of triangles in the generated mesh. Error metrics with a user-provided error threshold guide the triangle selection process. A survey of irregular mesh techniques is given in Heckbert and Garland [25].

Garland and Heckbert [21] describe a mesh refinement algorithm using Delaunay triangulation (Figure 2.3), which avoids thin sliver triangles by maximizing the minimum angle of all possible triangles from a given set of vertices. Such thin triangles contribute little to the resulting image and provide overhead when they affect less than a pixel in the final image. During each iteration of the algorithm, the vertex with the highest error level is selected. This vertex is added to the set of vertices, and the set is retriangulated. Cohen-Or *et al.* [8] divide the terrain into polygonal patches, where each patch has its own Delaunay triangulation. A top-down traversal of the resulting hierarchy allows for fast culling.

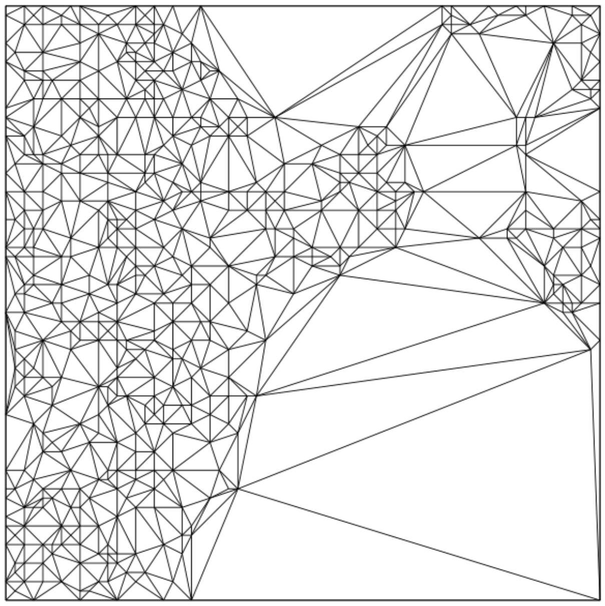


Figure 2.3: A Delaunay triangulation. [21]

Hoppe [28] adapts the progressive mesh representation of Hoppe [27] to terrain rendering. A progressive mesh encodes a continuous sequence of LODs in a hierarchy of refinement and decimation operations that operate on a small subset of triangles within the mesh. The refinement operations are vertex splits, where a vertex is replaced with two vertices. The decimation operations are edge collapses, where one vertex replaces two vertices that define an edge. Figure 2.4 demonstrates these operations.

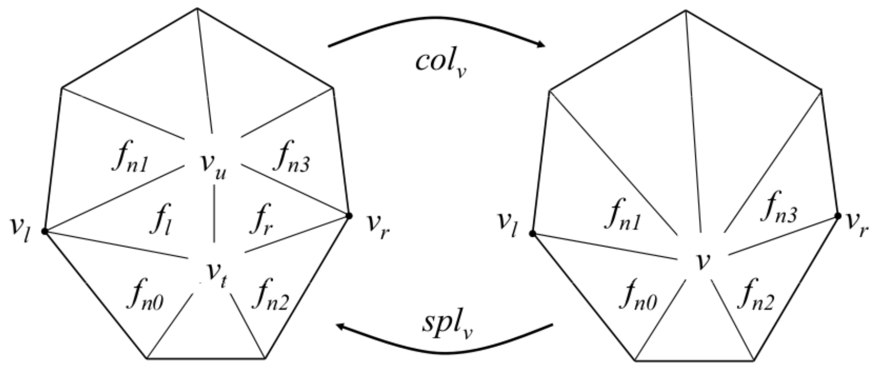


Figure 2.4: An edge collapse and a vertex split within a progressive mesh. [29]

Although these algorithms generate the fewest triangles that approximate the terrain within a specified error threshold, the unconstrained nature of the mesh places complex dependencies on the refinement of vertices. The nonuniform sampling of the terrain does not match the uniform sampling of the terrain provided by the raster data. Thus, techniques using irregular meshes have found more effective application representing arbitrary meshes [29]. Also, preprocessing prevents dynamic modification of the mesh at run time, thus composition and deformation are not possible with these techniques.

2.1.2 Semi-regular Meshes

Techniques that generate semi-regular meshes place more constraints on the size and arrangement of triangles than those that generate irregular meshes. These constraints simplify the data structures involved, reducing the amount of memory required. Regular subdivision is accomplished using an adaptive quadtree-based subdivision, as shown in Figure 2.5. The predictable structure of the generated mesh allows for the creation of triangle strips, which are more efficient for rendering. The structure also closely matches the regular nature of the input rasters, allowing for dynamic modifications to the height field at run time. A survey of this class of techniques is given by Pajarola and Gobbetti [53].

Von Herzen and Barr [68] introduce the restricted quadtree. The traversal of

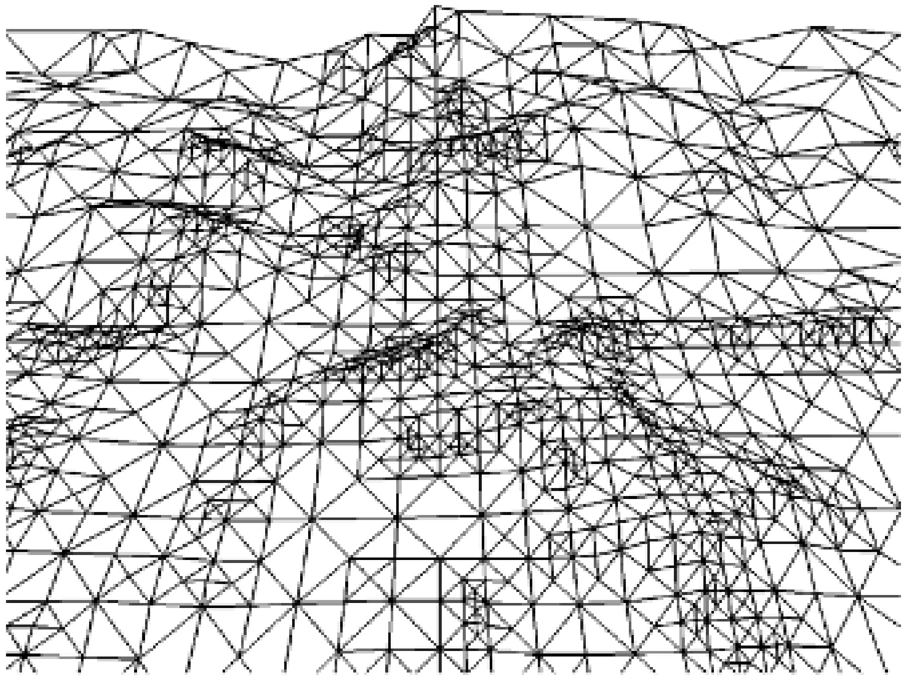


Figure 2.5: A quadtree-based subdivision results in a semi-regular mesh. [53]

a restricted quadtree operates with the additional constraint that neighboring nodes may differ by no more than one subdivision level. The triangulation of each node must be performed with respect to its neighbors. These requirements ensure that the generated mesh is conformant, meaning that it is smooth and without cracks. Figure 2.6 demonstrates a noncomforant triangulation, while Figure 2.7 shows a triangulation produced with these constraints.

Lindstrom *et al.* [36] propose the concept of operating on pairs of triangles within a restricted quadtree. The algorithm begins with the full-resolution mesh and merges triangle pairs to obtain a view-dependent triangulation of the terrain. A set of dependency rules are given to guide the algorithm in merge operations in order to produce conformant meshes. The dependency rules define for each vertex the vertices required for the input vertex to exist in the triangulation. These rules, shown in Figure 2.8, demonstrate that merging triangle pairs results in the same class of meshes as the restricted quadtree subdivision.

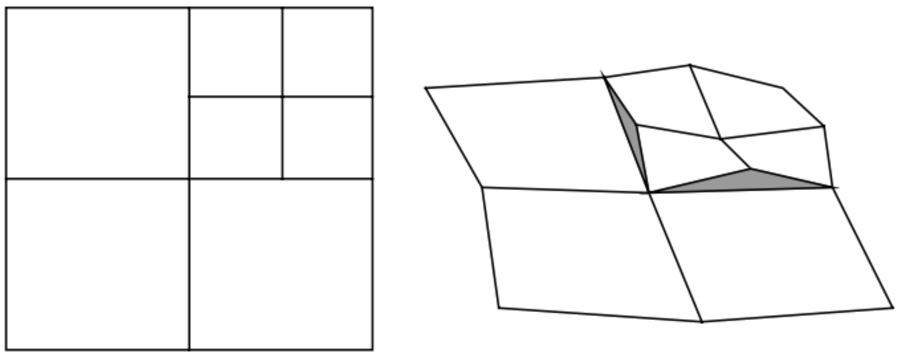


Figure 2.6: Cracks result from unrestricted quadtree subdivision. [53]

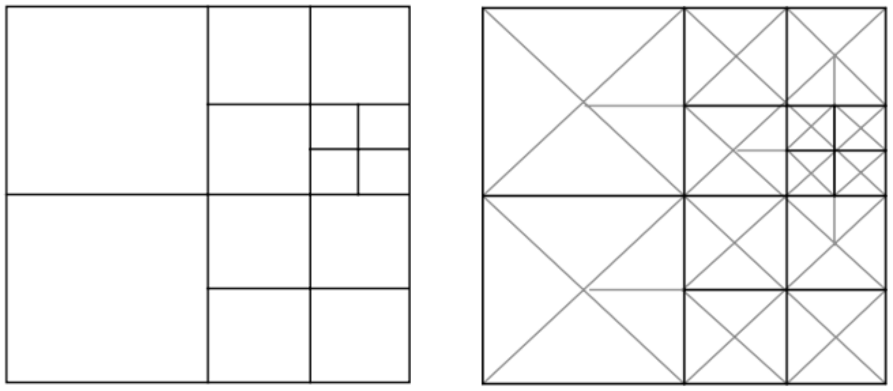


Figure 2.7: A smooth mesh results from restricted quadtree subdivision. [53]

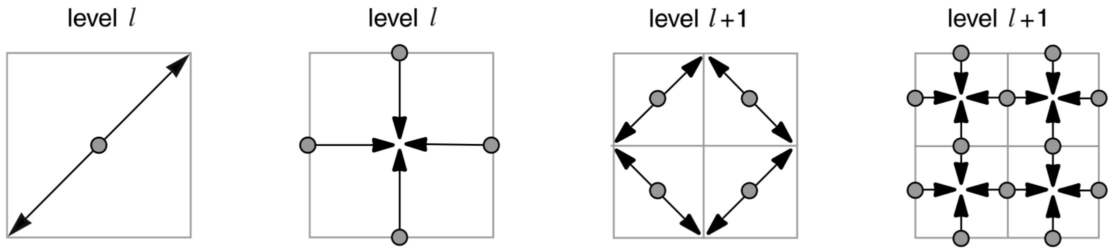


Figure 2.8: Dependency rules for vertices in a restricted quadtree. [53]

Triangle bintrees [6, 16] are related to restricted quadtrees and build upon the concept of operating on triangle pairs. Triangle bintrees build a binary hierarchy of triangles by subdividing a base triangle into two triangles using longest-edge bisection. The longest edge of the base node's triangle is divided into two edges, forming the edges of the two child nodes' triangles. The vertex added at the midpoint of the longest edge is known as the base vertex. A merge operation removes the base vertex, while a split operation adds the base vertex. Figure 2.9 demonstrates five levels of such a subdivision. Triangle bintrees require the same subdivision dependencies as Lindstrom *et al.* [36], implying that the two approaches produce the same class of meshes.

Duchaineau *et al.* [16] present Real-time Optimally Adapting Meshes (ROAM). The ROAM algorithm progressively refines and decimates a triangle bintree using a dual priority queue approach. One queue references the nodes with the least error and

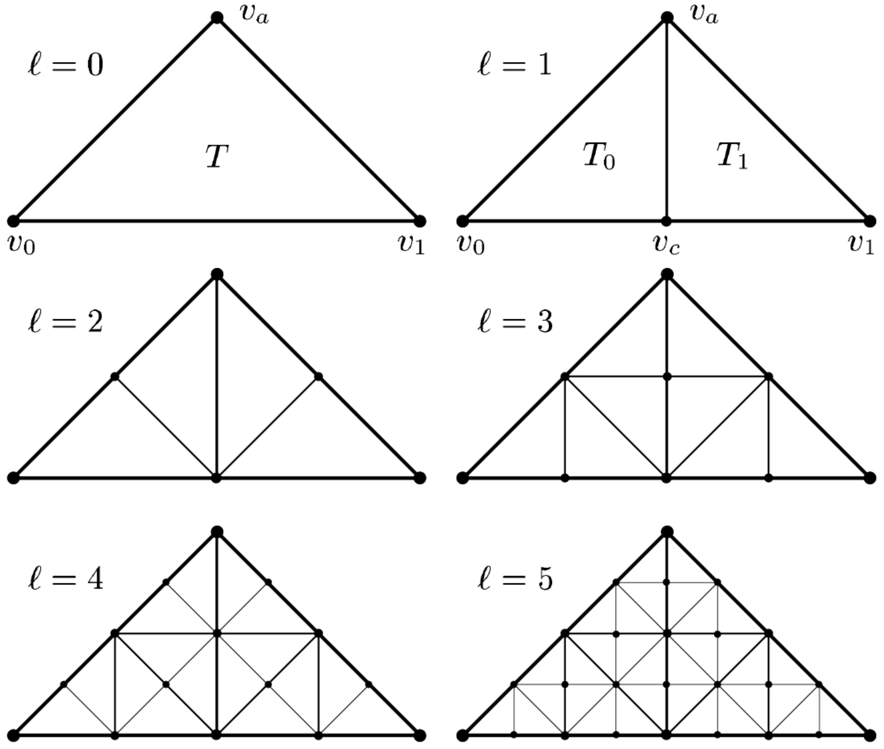


Figure 2.9: Five levels of a triangle bintree. [16]

the other queue references the nodes with the most error. As the viewer's position changes, the nodes with the least error are merged, and the nodes with the most error are split. This approach allows for progressive subdivision of the mesh, taking advantage of processing done for previous frames. The error values are based on a bounding volume called a wedgie. To determine a triangle's geometric distortion, its wedgie is projected into screen space and the maximum length of the wedgie's segments is stored. Figure 2.10 shows a ROAM triangulation from above, where the viewer is on the left looking right. Figure 2.11 demonstrates how this triangulation appears with respect to the viewer.

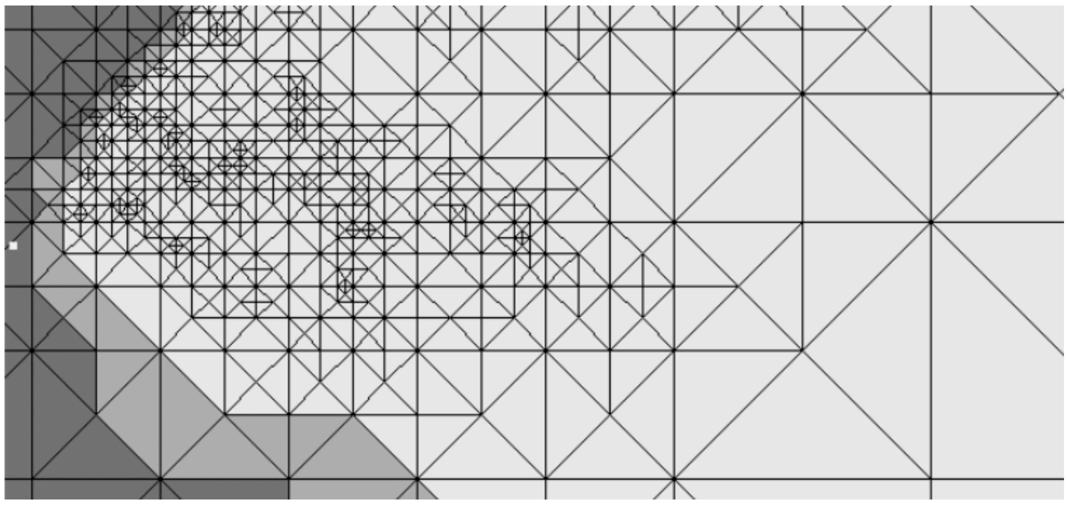


Figure 2.10: A triangulation of ROAM viewed from above. [16]

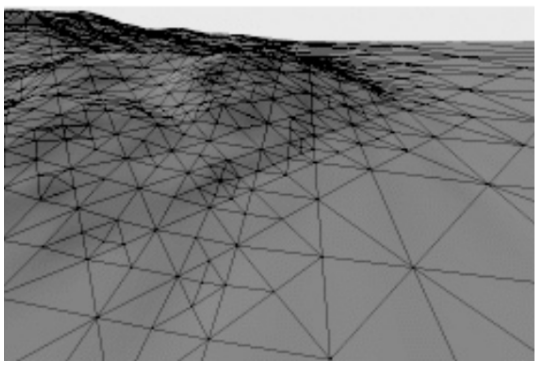


Figure 2.11: A triangulation of ROAM with respect to the viewer. [16]

Four performance enhancements that collectively reduce frame time by an order of magnitude are given by Duchaineau *et al.* [16]. They are view-frustum culling based on wedgie intersection with the view-frustum, creating triangle strips incrementally as splits and merges occur, deferring priority recomputation based on when it will effect a split or merge decision, and progressive optimization which is the ability to stop refinement once a maximum frame time is reached.

Pajarola [52] describes restricted quadtree triangulation, providing a number of optimizations for the restricted quadtree algorithm. Algorithms for refinement and decimation of a restricted quadtree are given. Pajarola also demonstrates how a restricted quadtree can be implicitly described, where the layout of the hierarchy in memory is contiguous and the node locations are defined mathematically. The implicit description greatly reduces the storage costs. Finally, the concept of representing a terrain using patches is described with respect to restricted quadtrees, a technique that proceeding algorithms build upon. Each patch is a restricted quadtree and can be independently refined or decimated. To maintain continuity between adjacent patches, adjacent vertices are morphed to match each other.

As commodity graphics hardware improved by having increased on-board memory and the ability to apply transformation and lighting, techniques that take advantage of these capabilities were developed. These techniques build off semi-regular triangulations.

2.2 Batched Approaches

The late 1990s marked a significant improvement in graphics hardware. With the release of the Nvidia GeForce 256 line in 1999 and the ATI R100 line in 2000, graphics systems gained the ability to offload the processing of vertices (transformation and lighting) from the CPU to the GPU. Triangle throughput increased, allowing more efficient strategies for generating a mesh.

Previous CPU-oriented algorithms optimize the number of triangles for graphics systems with a limited triangle budget. The algorithms spend a large amount of CPU time generating an optimal number of triangles. However, with increased triangle throughput, these algorithms do not use the graphics hardware to its potential, and the processing power of the GPU goes unused. Also, increased on-board memory and the limitations of CPU-to-GPU communication make retained mode graphics desirable. In retained mode rendering, mesh information is stored and rendered from GPU memory, as opposed to immediate mode rendering where the CPU sends data to the GPU every frame.

To address these issues, the following algorithms optimally feed the graphics hardware by processing large groups of triangles instead of individual triangles. These batched techniques split the terrain into patches of various extent and resolution. During preprocessing, detailed meshes for each patch are generated. At run time, visible patches are determined to a specific resolution and selected for rendering. Figure 2.12 shows a hierarchy of terrain patches [65].

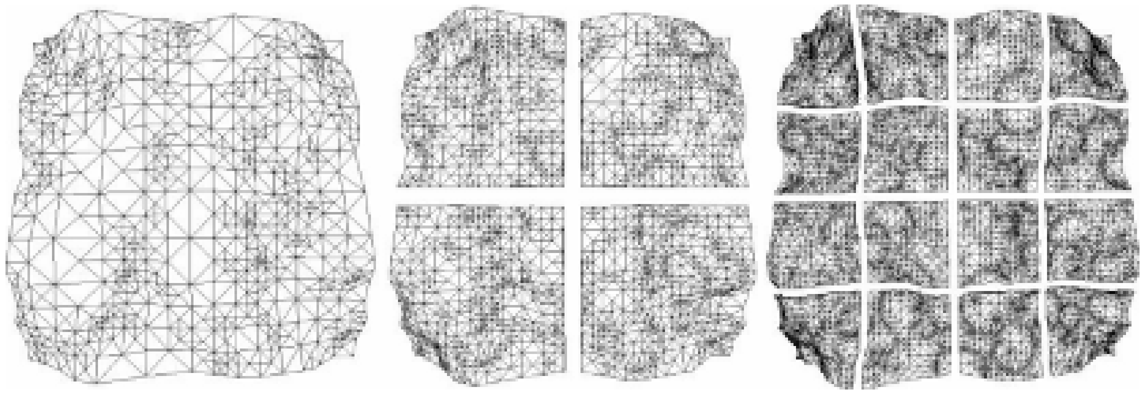


Figure 2.12: A hierarchy of terrain patches. [65]

The granularity tradeoff of operating on aggregate sets of triangles implies that more triangles must be rendered every frame. Despite an increased triangle count, batched techniques see an improvement in speed because there is less CPU processing occurring at run time to determine adequate levels of detail.

The challenge of these algorithms is to ensure that the borders between adjacent patches match to avoid the appearance of cracks in the generated mesh. Also, the effect of sudden transitions of patches between LODs as the view changes, called popping, must be mitigated.

Batched techniques are classified based on how patches are hierarchically managed. Techniques that use quadtrees of patches take advantage of the regular nature of raster data to generate the meshes for each patch at various LODs. Techniques that organize patches into triangle bintrees use more complicated preprocessing to generate high-quality meshes for each patch.

2.2.1 Batching Using Quadtrees

Approaches using quadtree subdivision for organizing patches take advantage of the regular nature of the raster data to generate patches. Geomipmapping [12] views geometry as a texture and applies mipmapping to the height data to produce a quadtree of patches during preprocessing. At run time, visibility culling is applied to determine which patches are inside of the view frustum. Patch LOD is selected based on distance from the viewer. This selection process is accelerated using error metrics created during preprocessing. To maintain a continuous mesh, vertices are collapsed, creating elongated triangles. Popping is removed by gradually morphing patch vertices between levels of detail as the viewer moves. Figure 2.13 demonstrates stitching a node with its neighbor along its left border. Figure 2.14 show how a vertex is morphed toward the triangle of a lower LOD patch.

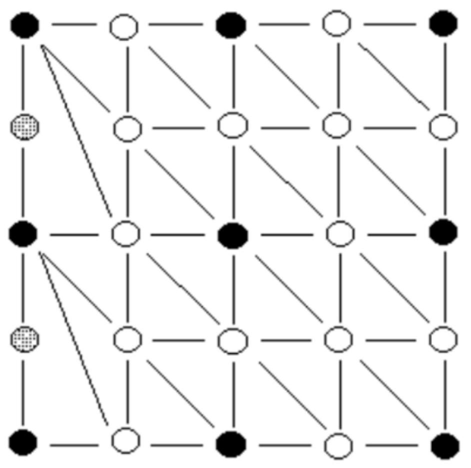


Figure 2.13: Vertices at patch borders are collapsed to maintain continuity. [12]

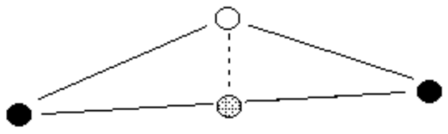


Figure 2.14: Patch vertices are interpolated between LODs to avoid popping. [12]

Chunked LOD [65] generalizes the approach of De Boer [12] and adds support for compression. Cracks between adjacent patches are hidden using skirts (Figure 2.15), which are meshes that extend vertically downward from the edges of patches. Skirts must extend to the bottom of the full-resolution mesh. Although simple, this method leads to texture stretching. Popping is handled as in De Boer [12] by using morphing. An example of a chunked LOD mesh is given in Figure 2.12.

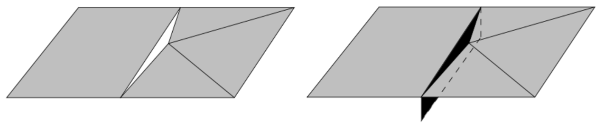


Figure 2.15: A vertical skirt hides cracks between patches. [65]

2.2.2 Batching Using Triangle Bintrees

ROAM Using Surface Triangle Clusters (RUSTIC) [56] uses the triangle bintree structure of the ROAM algorithm. In a preprocessing step, variable resolution patches are generated using the ROAM algorithm. To ensure a smooth mesh without cracks, the generated patches obey the edge constraint: adjacent tiles must have the same edge boundaries. At run time, these patches are managed as part of the ROAM algorithm. An analysis of the efficiency of the algorithm with respect to the size of each patch is given, showing that patch sizes of less than 32 triangles are inefficient. The use of the edge constraint limits the adaptivity of individual patches.

Batched Dynamic Adaptive Meshes (BDAM) [6] also use a triangle bintree to organize patches. Each patch, however, contains a high-resolution TIN created offline. The offline process can use any triangulation algorithm to maintain the quality of the input data; however, the greedy Delaunay insertion algorithm (similar to Garland and Heckbert [21]) is used. Figure 2.16 shows a triangulation created using the BDAM algorithm, demonstrating the semi-regular organization of patches that each contain an irregular mesh. BDAM is similar to RUSTIC, but it adds triangle stripping for efficient rendering of patches, as well as out-of-core streaming of large datasets.

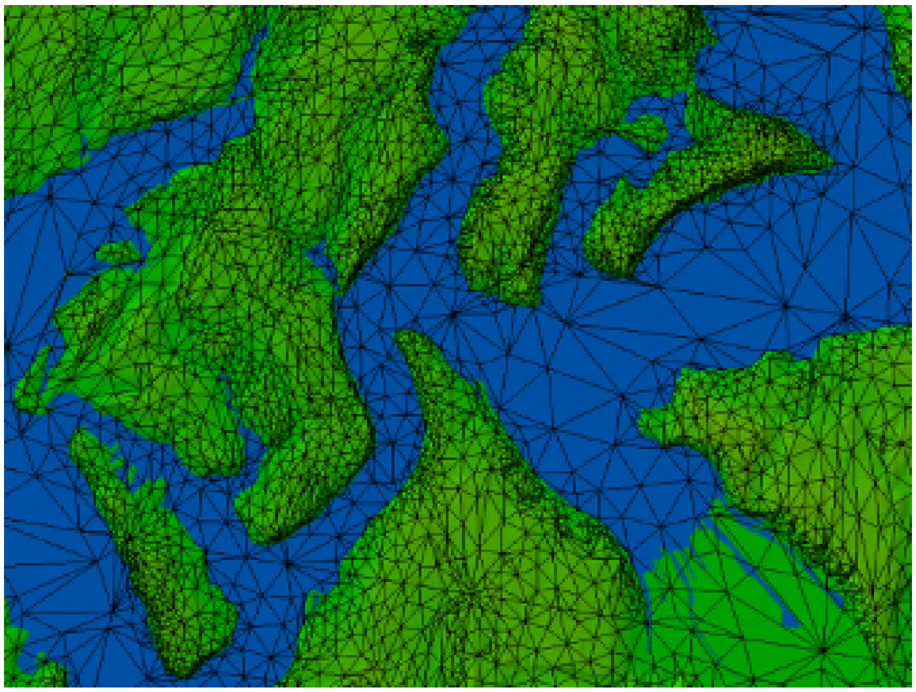


Figure 2.16: A triangulation created using BDAM. [6]

Planetary BDAM (P-BDAM) [5] adapts the BDAM algorithm for planetary rendering. Large terrain databases are managed and rendered using the batching capabilities of the GPU. The planet is divided into a number of tiles, where each tile corresponds to an individual triangle bintree. Accuracy issues presented by high-resolution planetary datasets are resolved using GPU hardware. P-BDAM uses a technique called "speculative prefetching," in which the areas of terrain needed for future frames are streamed into memory based on viewer movement. The preprocessing algorithm is distributed and designed to work with out-of-core datasets.

Batched techniques mark the transition from CPU-oriented algorithms to GPU-oriented algorithms. Although they take advantage of increased triangle throughput, these algorithms require lengthy preprocessing to generate meshes for each patch. Preprocessed complex data structures complicate run-time composition and deformation. Although the coarse-grained simplification of Lauritsen and Nielsen [35] has been adapted for terrain deformation by Brandstetter *et al.* [4], the technique has not been extended to work in the spherical domain required for planetary terrain rendering.

2.3 GPGPU Approaches

Until 2001, GPU functionality was fixed, meaning that the application programmer had little control over the GPU besides that provided by the graphics API. Graphics techniques that required functionality beyond what the hardware provided had to be implemented for the CPU or had to be implemented via API workarounds, if at all possible. The release of the Nvidia GeForce 3 and ATI R200 series of graphics cards expanded the possibilities for computing; these GPUs introduced programmability and opened the parallel processor, highly optimized for graphics rendering, to general purpose computation as well as advanced graphics techniques. The use of the GPU for such a task is called general purpose GPU (GPGPU). The concept of GPGPU computation can be applied even to graphics algorithms, where the GPU is used for computation closely related to rendering but not the actual rendering itself.

In the context of terrain rendering, algorithms can exploit programmability to generate real-time, view-dependent representations of the terrain on the GPU using on-board memory, freeing the CPU for time-critical tasks important to terrain rendering applications, such as streaming portions of terrain data from disk that cannot fit entirely in CPU or GPU memory. GPU-oriented approaches also open the way for advanced operations upon GPU-resident data, such as decoding of compressed terrain data [14, 39] and composition operations that allow for high-quality planetary rendering [33, 34].

2.3.1 Geometry Clipmapping

Geometry clipmaps [2, 39] build a view-dependent representation of the terrain at run time using GPU programmability. The texture clipmap [61], originally used for color textures, is adapted for height fields. Height data is prefiltered into a mipmap pyramid, where each successive level represents a version of the previous level at four times less resolution. At run time, nested rectangular grids are filled with data from the mipmap pyramid around the viewer's position, as shown in Figure 2.17. As the viewer moves, the location of the grids shift, and the resulting L-shaped regions are incrementally refilled. Previous data that remain within the bounds of the grids are left within, while data that has been shifted out of the grids are replaced with new data. This is possible using toroidal addressing of grid memory. Figure 2.18 demonstrates the shifting and update of data within the nested grids. Figure 2.19 shows how the nested grids appear with respect to the viewer.

As with batched techniques, the borders of adjacent clipmap regions must be smoothly interpolated. This is accomplished with programmable vertex shaders. While the original implementation in Losasso and Hoppe [39] takes advantage of programmability only for blending the area between grid levels, Asirvatham and Hoppe [2] describe a more GPU-intensive version of the algorithm.

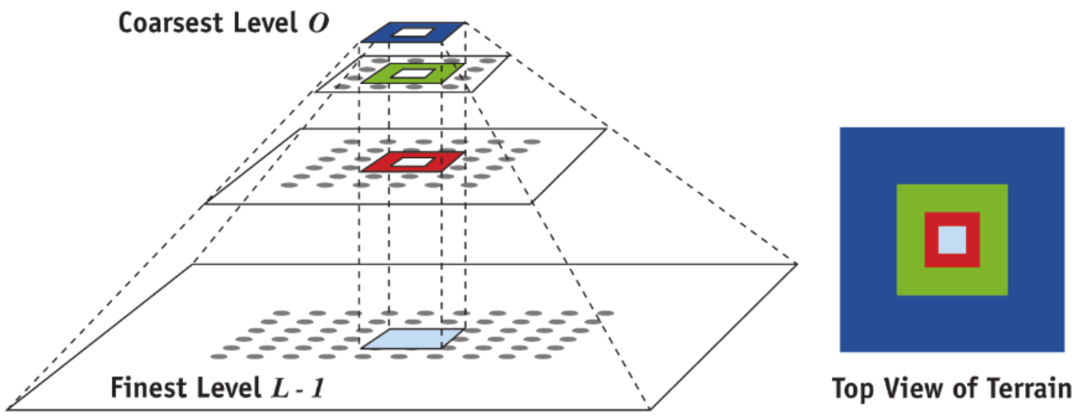


Figure 2.17: Nested rectangular grids filled from a mipmap hierarchy. [2]

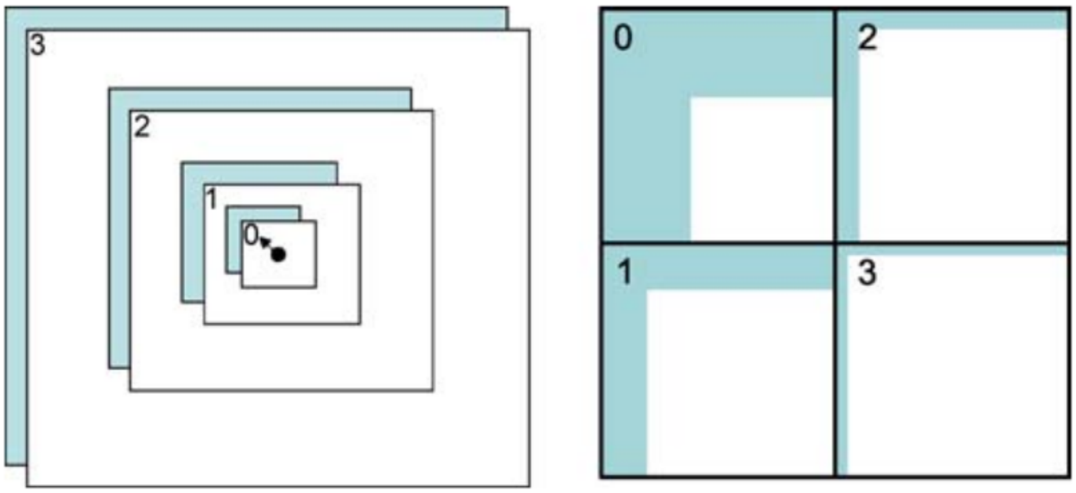


Figure 2.18: As the viewer moves, L-shaped grid regions are refilled. [37]

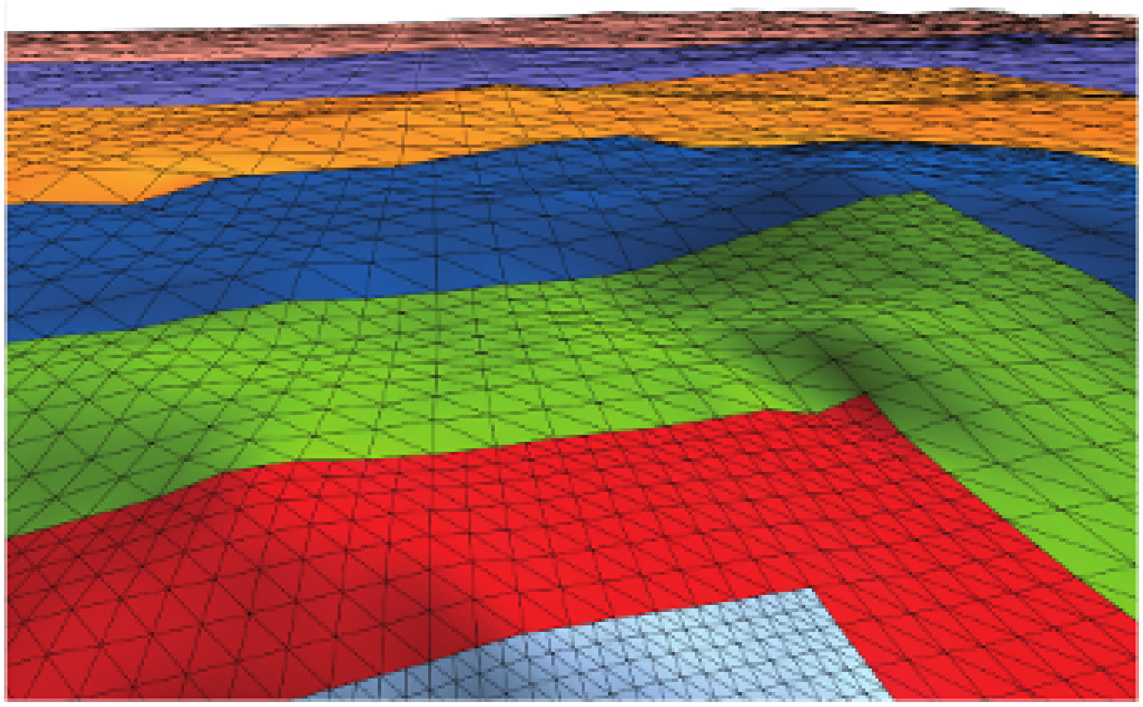


Figure 2.19: Nested grids rendered over the resulting terrain. [2]

Despite the use of programmability, data decompression occurs on the CPU, causing an eventual CPU bottleneck as the amount of data increases. Also, the rectangular grids do not map well to the spherical domain required for planetary rendering. Although Clasen and Hege [7] adapted geometry clipmaps for spherical rendering, no methods are given for reducing the distortion of spherically projected data.

2.3.2 Planetary-scale Terrain Composition

Kooima *et al.* [34] present a technique for rendering large planetary terrain databases using the GPU for most of the computation. The algorithm is oriented towards compositing many planetary datasets that use various map projections.

The first problem addressed is tesselating the surface of a sphere such that a uniform sampling is produced. Kooima [33] argues that a recursively subdivided cube, as used in Cignoni *et al.* [5], results in a nonuniform sampling of the spherical surface that leads to distortion at key vertices, as shown in Figure 2.20. Thus, the algorithm begins with an icosahedral subdivision of the sphere. Subdividing an icosahedron is shown to produce the maximum possible regularity for spherical subdivision, where most vertices have degree six. An example icosahedral subdivision is given in Figure 2.21. Icosahedral subdivision results in a sphere made up of spherical triangle patches, and a refinement algorithm similar to ROAM is used on the CPU to determine coarse-grained visibility and tessellation of the base mesh. The visible patches are then uploaded to the GPU for fine-grained refinement. On the GPU, each patch is subdivided the same number of times. However, patches resulting from the coarse-grained subdivision on the CPU are of various size, resulting in a mesh that remains view dependent.

After the view-dependent mesh is generated on the GPU, a number of composition operations involving each of the height maps used to describe the planet are applied to the positions in the mesh. The compositing operations are detailed in Section 2.4.2.

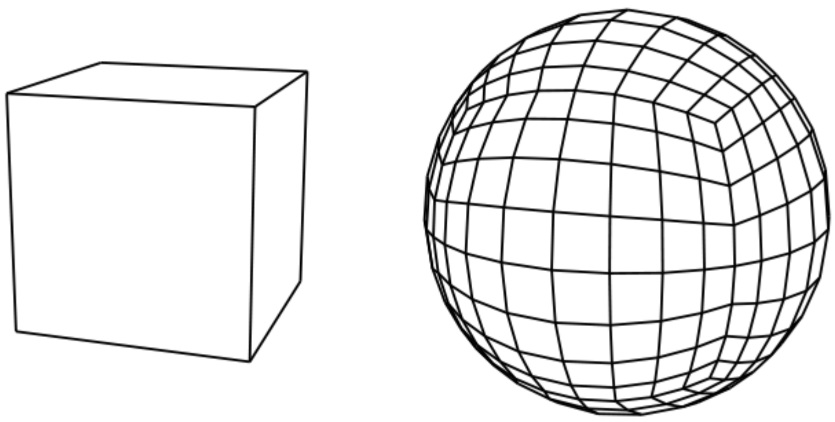


Figure 2.20: Subdividing a cube to approximate a sphere leads to distortion. [33]

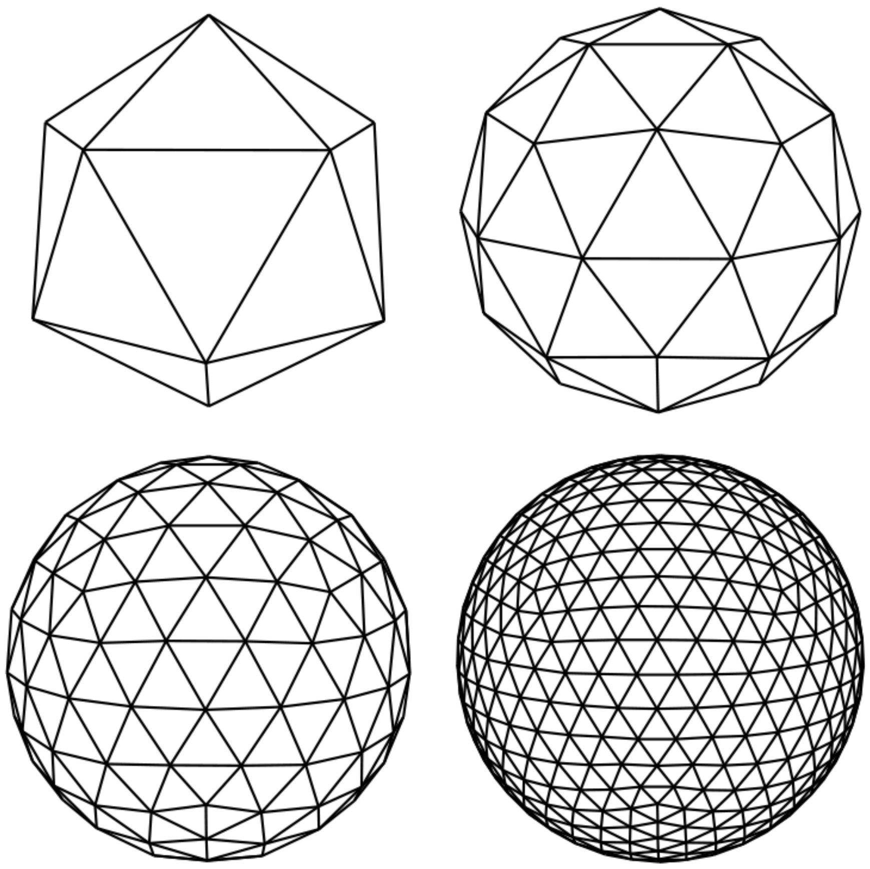


Figure 2.21: Subdividing an icosahedron minimizes distortion. [33]

The approach of Kooima [33] is unique in that it includes a tessellation algorithm specifically for spherical terrain and it demonstrates that advanced composition algorithms are possible when data for the terrain resides entirely within color buffers, whether this data is geometry or photographic imagery of the planet surface. CPU processing is necessary for geometry generation, although this processing is described as having a minimal effect on efficiency.

2.3.3 Ray Casting

Ray casting of height fields has been performed on the GPU in the form of relief mapping [49] and parallax occlusion mapping [62] among other techniques. The advantage of ray casting is that the complexity of the algorithm scales with the size of the framebuffer rather than the size of the dataset. Assuming the ray-height field intersection algorithm finds the correct intersection point, an accurate, view-dependent image is generated, obviating the need for mesh generation and the preprocessing of complex hierarchies. Ray casting for surface detail lies within the class of algorithms known as displacement mapping, and a survey of such algorithms is given in Szirmay-Kalos and Umenhoffer [60].

The general approach is to cast a ray in a fragment shader for each pixel that the displaced surface could possibly affect. If the ray intersects the height field, the intersection point, described by texture coordinates, is used to query color and normal maps for the surface. Because a ray could intersect the height field at multiple points (Figure 2.22), most algorithms move along the ray in various ways to find the first intersection.

Besides being computationally intensive, this approach presents a problem when rays enter the volume of the height field at low, grazing angles. Horizontal rays parallel to the height field maximize the amount of height field that could be erroneously skipped, leading to artifacts. Figure 2.23 demonstrates artifacts resulting from rays missing the first intersection, visible along the visible edges of the surface, and artifacts resulting from large step sizes that cause early intersections, visible within the surface.

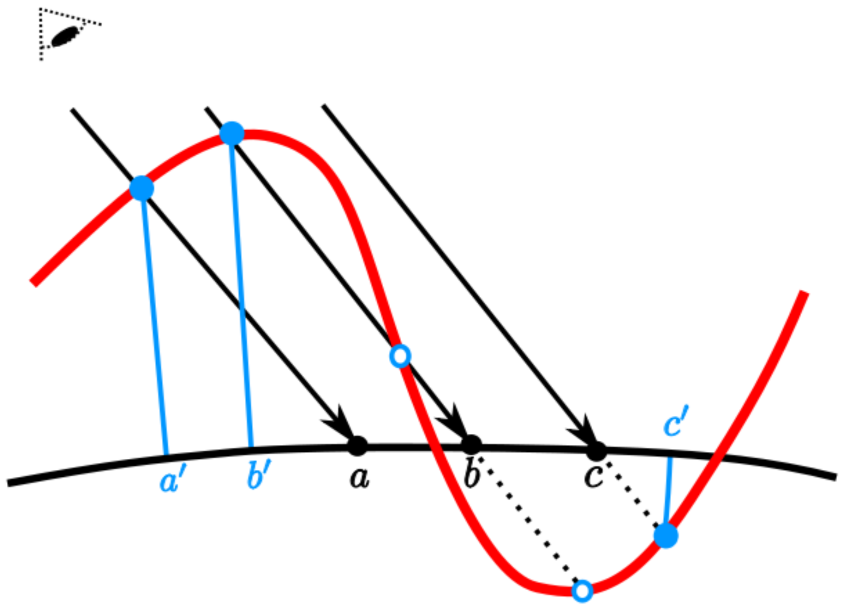


Figure 2.22: A ray may intersect the height field at multiple points. [19]

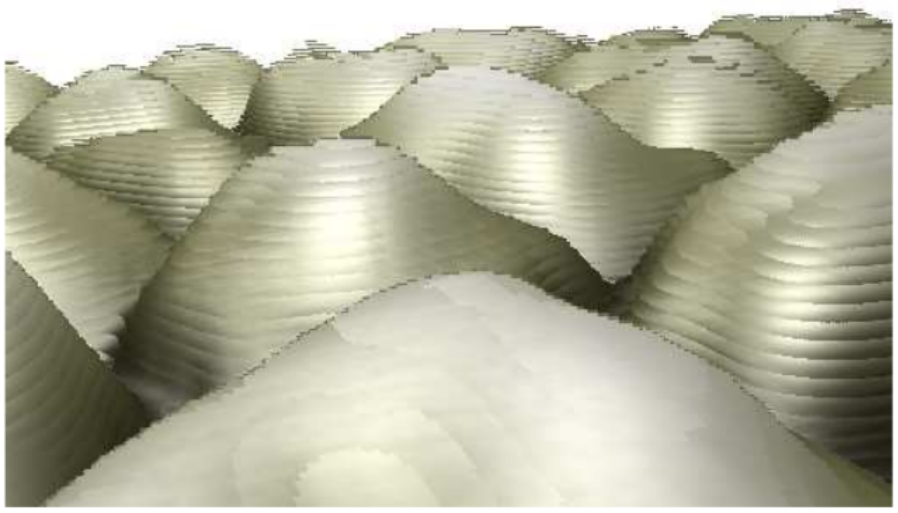


Figure 2.23: Incorrect ray intersections produce artifacts. [49]

Numerous acceleration techniques for finding the first ray hit point have been developed [15, 17, 55, 63], but most require non-trivial preprocessing of the height field. However, Tevs *et al.* [63] show that maximum mipmaps of the height field are a simple acceleration structure that results in accurate intersections. Maximum mipmaps can be efficiently generated on the GPU.

Dick *et al.* [13] demonstrate an algorithm that ray casts large-scale terrain data using maximum mipmaps. A tile-based hierarchy of height field patches is used to apply visibility culling before the ray casting operation. For each visible tile, the back faces of the tile's bounding box are rasterized to instantiate the fragment shader for each possible pixel that the tile affects. Ray casting proceeds using the maximum mipmap of the tile, which is computed during the preprocessing stage. The ray casting algorithm is compared with a rasterizer using the same tile-based hierarchy, and it is shown that rasterization performs better than ray casting at moderate height field resolutions. However, ray casting provides more consistent framerates and performs better for large height field resolutions.

The results of Dick *et al.* [13] show that terrain ray casting could be effectively applied to large planetary-scale datasets. The application of ray casting to planetary data involves reformulating the ray casting algorithm for the spherical domain. A difficulty of ray casting a sphere for high-resolution planetary data lies in floating point precision issues, as described in Kooima [33]. Ray casting of a displaced sphere requires referencing the radius of the sphere. However, for planetary data the radius is a number of an order of magnitude that uses nearly all the precision of a single-precision floating point value. Computations involving such a number reduce the available precision for high resolution data down to a meter close to the viewer.

2.3.4 Projective Grid Mapping

The advantages of both ray casting and rasterization motivate the use of projective grid mapping [37, 38]. In comparison to ray casting, rasterizing a mesh takes advantage of the graphics hardware, which is designed and optimized for rasterization.

However, the advantage of ray casting is the creation of a view-dependent image. PGM attempts to combine the benefits of both approaches by generating a mesh using ray casting, and the mesh is then rasterized. The ray casting operation can be implemented on the GPU. Although Hu *et al.* [29] generate a mesh on the GPU using the progressive mesh algorithm, the technique requires complicated traversal and management of data structures in parallel and is most effective for arbitrary surfaces.

The algorithm originated in wave animation [26, 31]. It begins by creating a regular grid that extends across the camera's view plane, where the points on the grid constitute a regular mesh as in Figure 2.1. Livny *et al.* [37] refer to this grid as the "persistent grid" owing to its static resolution, however the term "projective grid" is also used within the literature [38]. Figure 2.24 shows how the projective grid appears on the view plane. The grid points are two-dimensional vertices that exist in clip space, where the x and y values range from -1 to 1.

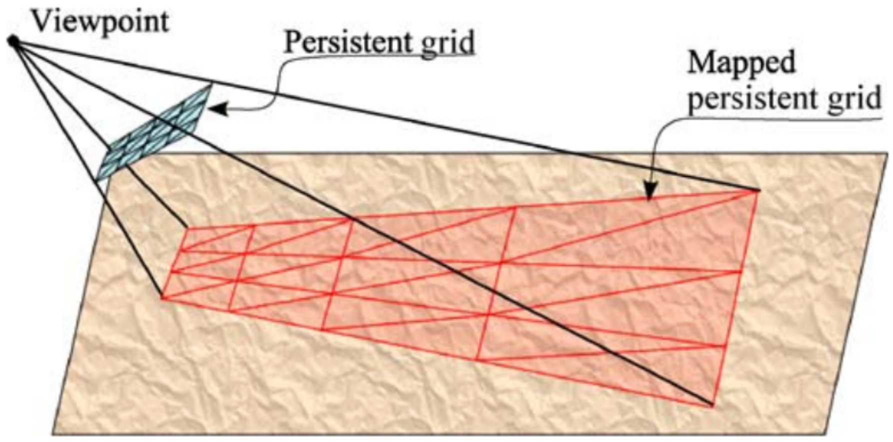


Figure 2.24: The grid is projected onto the terrain base plane. [37]

During rendering, a ray is cast from the view point through each grid point to the base plane of the displaced surface (usually $z = 0$). For this operation, the inverse projection of the grid point's clip space position is used to create a ray in world space. Where each ray intersects the base plane, the height value is obtained by querying the texture. The intersection point is displaced along the normal of the base plane by the height value and used as the position of the corresponding grid vertex in the mesh. Figure 2.25 illustrates this process. The projection of the grid onto the terrain base plane in Figure 2.24 demonstrates how this mapping leads to an adaptive sampling of the terrain where close regions of the terrain are sampled at a higher frequency than far regions.

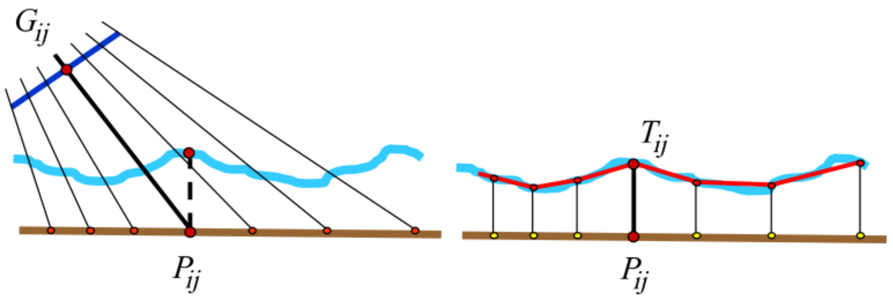


Figure 2.25: Mapping of grid points to terrain positions. [37]

Projecting a grid has various advantages [57]. The projection of each grid point is independent of the other grid points and can be performed in parallel on the GPU. In addition, the grid can be cached in GPU memory for efficient retained mode rendering. Triangles project to approximately the same area in screen space. If the projected grid is sized to the viewport, then each triangle is approximately pixel sized. This is the optimal scenario described by Pajarola and Gobbetti [53]. The number of triangles rendered is consistent, allowing the mesh resolution to be resized for the specific hardware. Also, the constant size of the mesh leads to steady frame rates.

The technique also presents a number of challenges. The computation of vertex normals required for lighting must be performed each frame as the mesh changes. This is not a problem if a normal map generated from the height map is available, because the normals of the original height field can be applied to the mesh instead of the normals produced from the mesh, which is only an approximation. Projected grid points that do not intersect the terrain base plane are invalid and, if not handled correctly, cause artifacts such as incorrect intersection points. In addition, the area below the view frustum may need to be sampled due to peaks that could intersect the frustum, as shown in Figure 2.26. If the maximum height of the terrain is known beforehand, a sampling camera separate from the viewing camera is defined such that the sampling camera contains only those parts of the terrain that could possibly affect the viewing camera [31]. Finally, at low horizontal angles the sampling of distant features is extremely sparse, and features could be missed, as shown in Figure 2.27. In addition, as the camera moves, the already undersampled regions of the terrain become resampled and a wave-like effect appears. This is not a problem for wave animation where rippling effects in the surface are expected. However, it is a problem for terrain rendering.

Livny *et al.* [37] apply projected grids to terrain rendering and address their drawbacks. A sampling camera shares the view position of the viewing camera, but it is given an orientation and field of view such that it samples only the area of the base plane that could possibly affect the viewing camera when imaged. This is

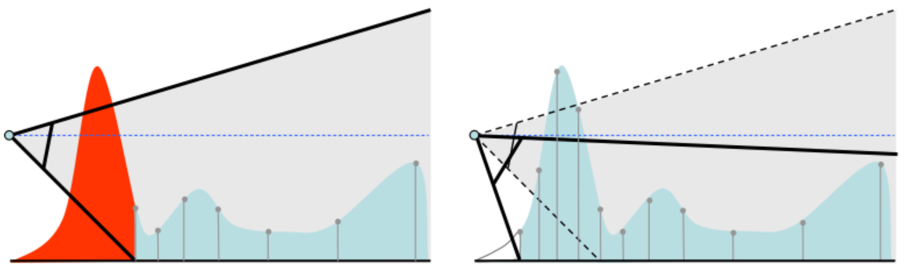


Figure 2.26: The area under the camera must be sampled. [37]

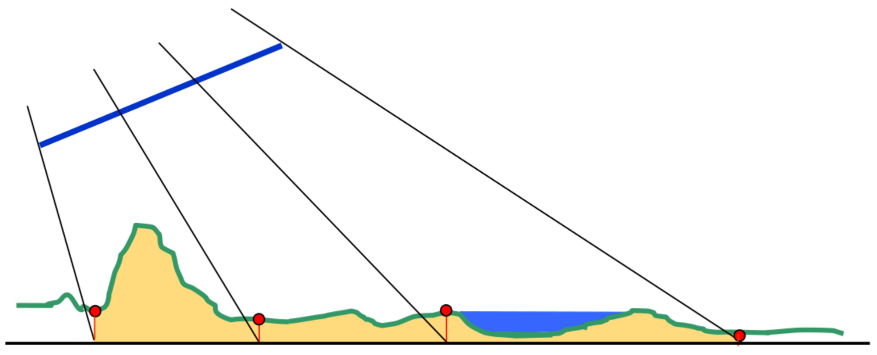


Figure 2.27: Details missed due to undersampling. [37]

accomplished using a maximum mipmap of the height data. The maximum mipmap allows the maximum height beneath the camera to be known. To mitigate the effects of undersampling for areas far from the viewer, the vertex shader selects an LOD from a mipmap hierarchy constructed with the height map. The higher levels in the mipmap exhibit a lower frequency, reducing the effects of sparse sampling.

Loffler *et al.* [38] extend PGM in an attempt to better account for undersampling. The importance-based warping method of Dachsbacher and Stamminger [11] is used to shift the horizontal rows of the projected grid such that the sampling of the terrain base plane is uniform. During the projection of the grid points, an object-space error measure for each grid point is calculated and projected to screen space, resulting in a rasterized error buffer along with the rasterized mesh. For any pixel with a corresponding error value in the error buffer that is above a user-defined threshold, a ray casting operation is performed to obtain an accurate color value for the pixel.

Using PGM to produce acceleration information for ray casting is a unique approach. However, the importance-based warping that occurs before projection of the grid must be done on the CPU, and uniformly sampling the terrain base plane counteracts the goal of the projected grid, which is to adaptively sample the terrain, as shown in Figure 2.24.

Schneider *et al.* [57] apply the projected grid technique to the spherical domain for procedurally generated terrain. The grid is projected onto a plane, where each point on the plane is mapped to an angular position on the sphere. The occluded area of the sphere that could possibly rise above the horizon is not accounted for, and Schneider *et al.* [57] notes that the use of fog can obscure these missing details. The use of the projected grid in this way is sufficient for a procedural terrain generation application where fidelity to real-world datasets and accurate visualization is not necessary.

To effectively apply PGM to the spherical domain, three previously unadressed issues must be solved. First, a scheme must be invented for sampling the area beneath the viewing camera on the sphere that could possibly affect the viewing camera.

Second, a strategy must be developed for sampling the portion of the occluded side of the planet that could rise above the horizon and affect the image. Finally, a reference sphere should be chosen so that the projected mesh, when displaced, does not result in large triangles. An appropriate reference sphere leads to approximately pixel-sized triangles when the grid points are ray cast and displaced from the sphere.

2.4 Planetary Terrain Rendering

Correct planetary rendering requires attention to various issues. Kooima *et al.* [34] address many aspects of planetary rendering that are not considered in terrain algorithms that focus on planar terrain. The map projections used affect how datasets should be mapped to the sphere and how they should be sampled. The composition operations described by Kooima *et al.* [34] remove the effects of distortion at the poles and provide an important functionality for terrain renderers intended for use in scientific visualization. Finding sources of planetary data suitable for the application is another problem for rendering planets.

2.4.1 Map Projections

A common problem of planetary terrain renderers is distortion at the poles [33]. Figure 2.28 shows a screenshot from Google Earth 5.1 [24] taken of Mars's south pole, which demonstrates visible distortion in the form of lines of latitude converging at the pole. This complication is partially a result of the map projection used to organize data in the image.

Geographic information is transformed onto the 2D plane of an image by a process known as map projection [32]. Specifically, a map projection entails two sets of equations that transform between geographic coordinates (λ, ϕ) to cartesian coordinates (x, y) of the map projection, henceforth called projection coordinates. The forward mapping equations transform geographic coordinates to projection coordinates, and the inverse mapping equations constitute the inverse transformation. The forward mapping equations are more useful in planetary rendering since each vertex

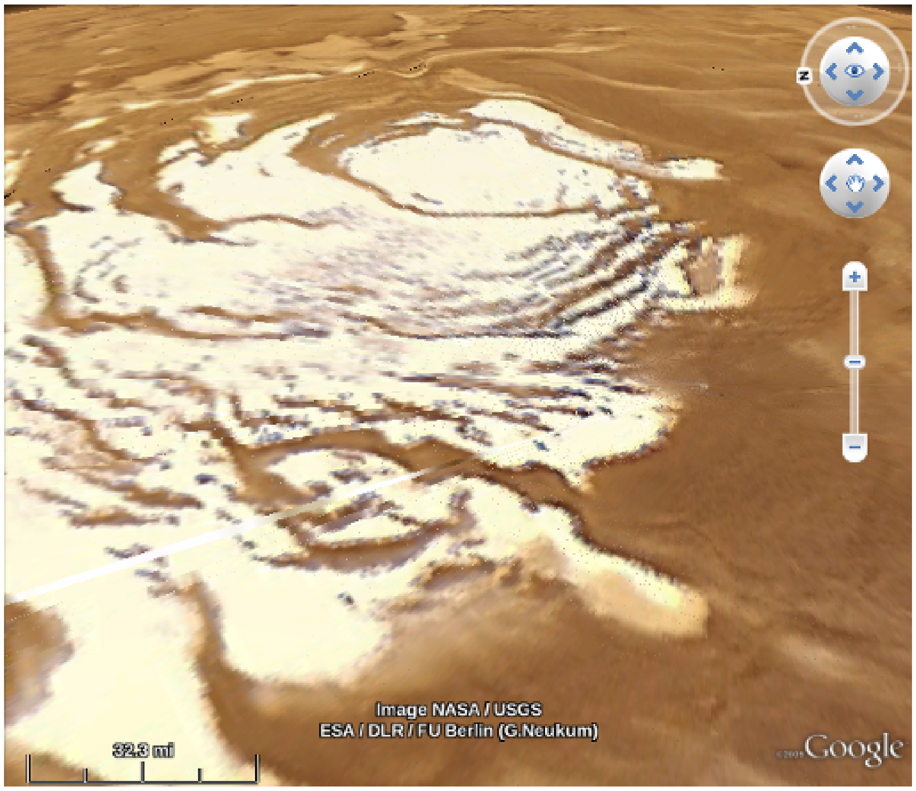


Figure 2.28: An image of Mars's south pole produced by Google Earth. [24]

in the generated mesh owns a pair of geographic coordinates, and we want to transfer this to coordinates of the images that we want to overlay onto the planet.

Graphical modeling necessitates a further transformation from projection coordinates (x, y) to texture coordinates (s, t) or pixel coordinates (i, j) of the image. Although planetary data sources include documentation on the map projections used, equations for obtaining texture coordinates are not given because this is a domain-specific issue. Our method of transforming projection coordinates to texture coordinates is given in Section 4.1.

A map projection can distort geographic information with respect to any of the following criteria [32]:

- An equal-area map projection preserves the surface area of the reference surface.
- An equidistant map projection preserves the length of lines on the reference

surface.

- A conformal map projection preserves the angles between lines on the reference surface.

The center of projection (λ_0, ϕ_0) is the origin of projection coordinates and the location where the scale of the map is true and the distortion is minimized.

There are many map projections, but there are two that are commonly used for visualization. The equations for these two are detailed by Eliason [18]. The equirectangular projection maps longitude and latitude directly to projection coordinates (x, y), as demonstrated in Equations 2.1 and 2.2. R is the radius of the reference ellipsoid at the center of projection. Owing to the simplicity of these equations, the equirectangular projection is preferred for reasons of efficiency and precision. In addition, the equirectangular projection shows little distortion near the equator; however, much distortion exists near the poles. A special case occurs when the latitude of the center of projection $\phi_0 = 0$ and is referred to as the simple cylindrical or Plate Carrée projection.

$$x = R\,(\lambda - \lambda_0)\,\cos(\phi_0) \tag{2.1}$$

$$y = R\,\phi \tag{2.2}$$

The distortion in this map projection is a combination of two causes. Because the equirectangular projection mostly preserves area at the equator and poorly preserves area at the poles, the pixels of the image have a rectangular shape at the equator and a deformed shape at the poles, as shown in Figure 2.29. The nature of this projection is that the middle line of pixels in the image describe the entire equator, while the top line of pixels describe the North Pole, a comparatively small region of the planet. Thus it is important to note that although distortion exists in the image, a relatively abundant amount of data about the poles exists in the equirectangular projected image.

The second cause of the distortion problem is the use of rectangular filters for mipmapping. Because of the nature of mipmap construction, mipmapping uses a

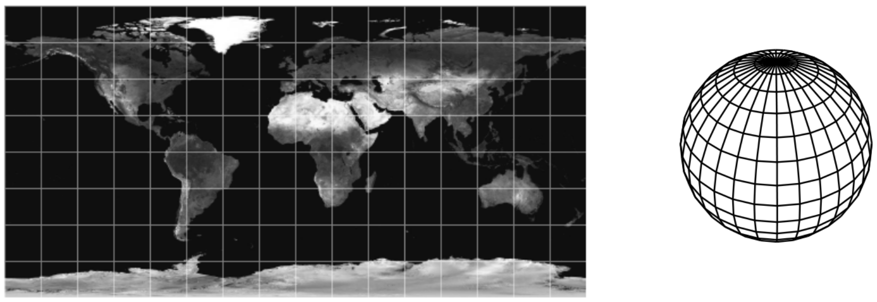


Figure 2.29: Equirectangular projected data mapped to the sphere. [33]

rectangular filter for averaging areas of the image. For equirectangular projected data near the equator, this filter closely fits the shape of each pixel. Near the poles, however, the pixels assume a deformed shape, and the rectangular filter no longer approximates the shape of the pixels. The result is incorrect filtering as demonstrated in Figure 2.30. The same polar region of Mars is shown without mipmapping in Figure 2.31.

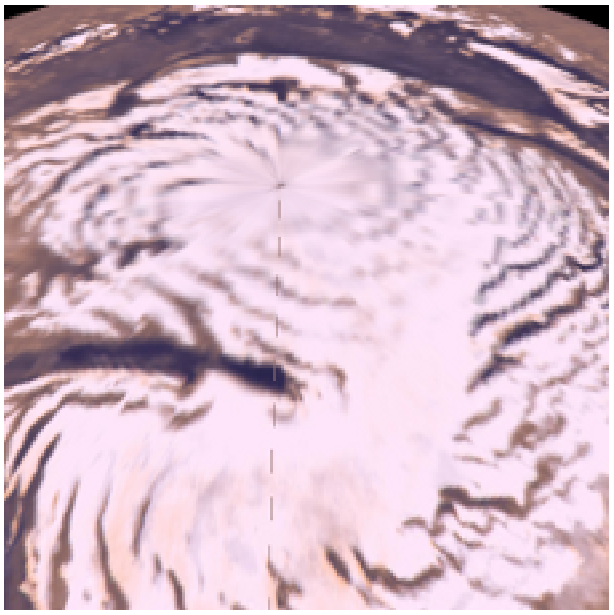


Figure 2.30: Mars's north pole displayed with mipmapping.

The solution is to use data in a map projection that sufficiently minimizes dis-

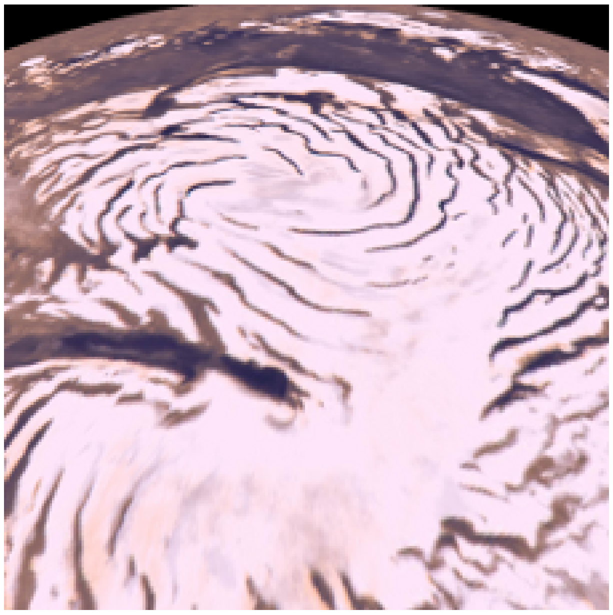


Figure 2.31: Mars's north pole displayed without mipmapping.

tortion in the covered area. For this purpose, the polar stereographic projection approximates polar regions well. Equations 2.3 and 2.4 represent the forward mapping equations [18]. R_p is the polar radius of the reference ellipsoid. Unlike the equirectangular projection, the polar stereographic projection results in a rectangular shape of pixels around the center of projection. An example of a polar stereographic projection is shown in Figure 2.32.

$$x = 2\,R_p\,\tan(\frac{\pi}{4} - \frac{\phi}{2})\,\sin(\lambda - \lambda_0) \tag{2.3}$$

$$y = -2\,R_p\,\tan(\frac{\pi}{4} - \frac{\phi}{2})\,\cos(\lambda - \lambda_0) \tag{2.4}$$

As a general guideline, regions within ±60 degrees latitude are mapped using the equirectangular projection, and regions above or below these parallels are mapped using the polar stereographic projection. With these map projections Kooima [33] implements data composition, described next, which allows the use of multiple datasets in differing map projections.

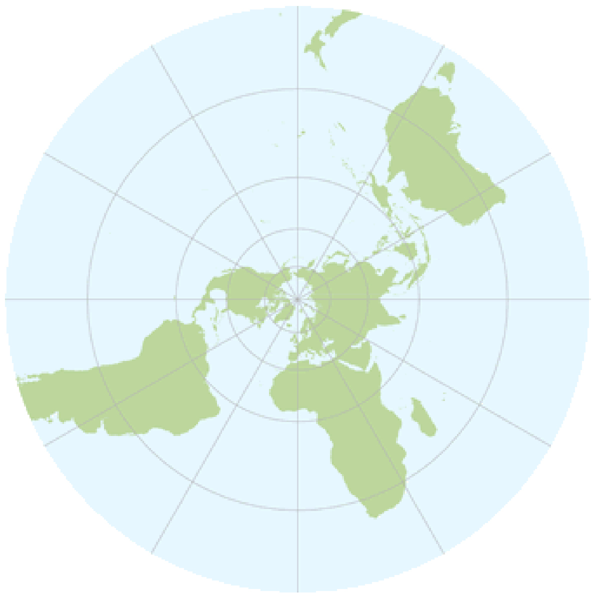


Figure 2.32: A polar stereographic projection. [32]

2.4.2 Data Composition

Kooima *et al.* [34] present a GPU-based technique for combining overlapping datasets at run time. Datasets are composited via deferred texturing, a process similar to deferred shading.

Deferred shading is a rendering technique that avoids lighting calculations for fragments that are ultimately discarded due to failing the depth test [54]. Objects in the scene are rasterized, and information such as the normal, color, specular, and depth for each fragment are written into as many buffers. These buffers are collectively termed the G-buffer and describe the visible surfaces in the scene. Figure 2.33 shows the buffers that make up the G-buffer. From the top left are the normal, color, specular, and depth buffers. In the last step, for each light source, lighting calculations are applied over the pixels in the G-buffer to create the final image in Figure 2.34.

The situation is similar in deferred texturing for planetary rendering: multiple compositing operations, instead of multiple lighting operations, are applied. Before the height compositing operations, the buffers describing the terrain mesh are gen-

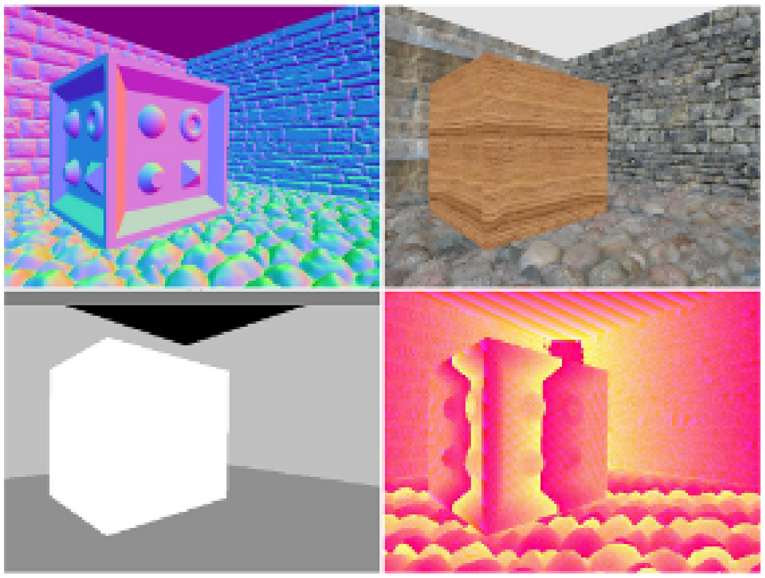


Figure 2.33: Various buffers used for deferred shading. [54]

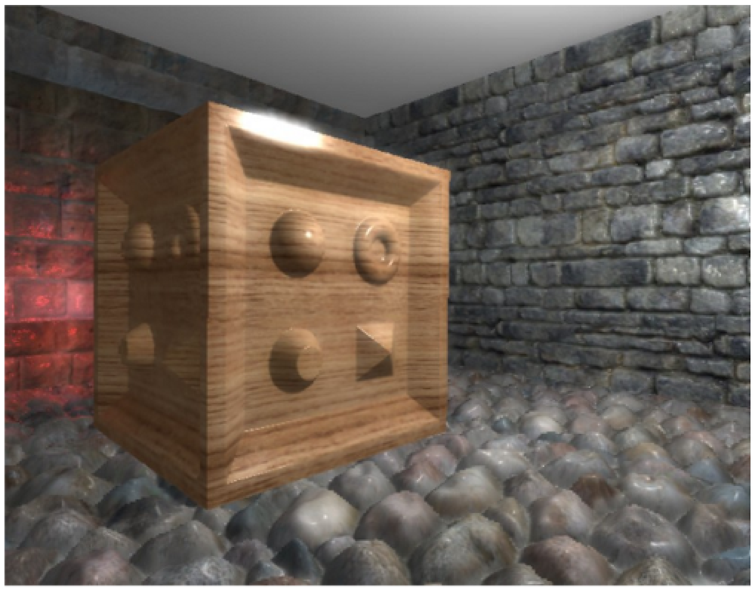


Figure 2.34: The result of deferred shading. [54]

erated using a GPU mesh refinement algorithm. The information in these buffers includes a set of vertices that constitute the mesh, a set of sphere normals for each vertex, and a set of geographic coordinates for each vertex. The vertices in the mesh are positions on the reference sphere that have yet to be offset by values in the height maps.

Height composition proceeds by executing a composition operation for each height map. A single step begins by binding the current height map and rendering a quad over the mesh texture. For each fragment in the framebuffer, the geographic coordinates corresponding to the fragment are used to read a value from the height texture. The corresponding vertex is offset by the sphere normal scaled by the height value. The resulting mesh and supporting information is rasterized into screen-sized buffers, creating the equivalent of a G-buffer in deferred shading. The color and normal composition operations take place in screen space. Instead of offsetting vertices as in height composition, the accumulated results of color and normal composition are written to a pair of accumulation buffers.

Because a texture cannot be read from and written to at the same time, two accumulation buffers are used. During each step of the algorithm, one of these buffers is bound for reading, and the other is bound for writing. At the end of a step, the buffers swap roles: the write buffer becomes the read buffer, and the read buffer becomes the write buffer. The use of two buffers in this manner is referred to as "ping ponging." Once the colors and normals have been composited, the algorithm applies lighting calculations using each pixel from the final color and normal buffers.

The entire process is illustrated in Figure 2.35. From the top and left to right, the buffers shown are the generated mesh, the mesh after height composition, the rasterized geographic coordinates, the accumulated surface normals, the accumulated colors, and the final image with atmosphere and lighting applied. In the first two images, the buffers are rendered for the purpose of demonstration. Also, the values in the third image are scaled by a factor of 100 to make the granularity of the geographic coordinates more apparent.

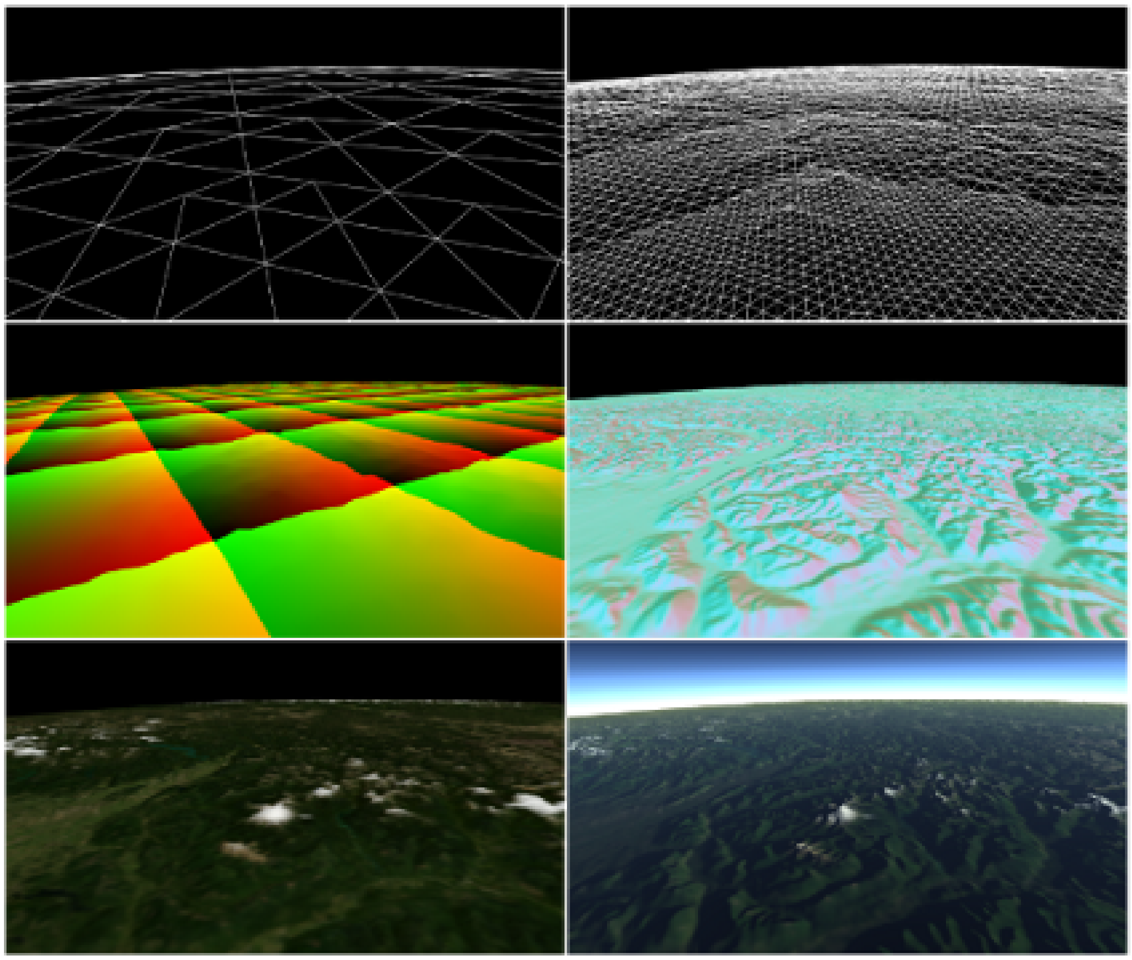


Figure 2.35: Various buffers used for data composition. [33]

Although the results of composition could be saved and used across multiple frames, composition is performed each frame due to the restriction that geometry must be generated and operated upon in eye space. Working in eye space overcomes the floating point precision issues caused by using world-space coordinates with the center of the planet as the origin.

Finally, to ensure that distorted pixels do not affect the final image, a weighting factor is calculated. The weighting factor is used to mix the incoming height values and the currently accumulated height values.

2.4.3 Data Sources

A vast amount of data exists for many planetary bodies and is freely available on the Internet. In particular, NASA's Planetary Data System (PDS) [42] indexes data from all past and present missions in an attempt to ensure the longevity of NASA data. Each dataset from NASA comes with a PDS label that describes the map projection used, geographic extents, and other useful information. Furthermore, the Geospatial Data Abstraction Library (GDAL) [23] and associated command-line utilities are able to read PDS labels, simplifying data processing.

The planetary bodies that are possibly of most interest are Earth, the Moon, and Mars, and there a number of NASA missions that have produced data using the equirectangular projection and the polar stereographic projection. NASA missions for Earth include the Shuttle Radar Topography Mission (SRTM) [44] mapped elevation data for almost the entire area of Earth between ±60 degrees latitude. Landsat Image Mosaic of Antarctica (LIMA) [41] and the MODIS Mosaic of Antarctica (MOA) [47] are mapped using polar stereographic projections.

The primary test case for this work is Mars. The Mars Orbiter Laser Altimeter (MOLA) [43] data maps the topography of Mars and provides data for both global and polar data. The MOLA instrument is a device on the currently nonoperational Mars Global Surveyor (MGS) satellite. MOLA provides global height maps of the areoid, a model of the general shape of Mars determined by the overall gravitational field of

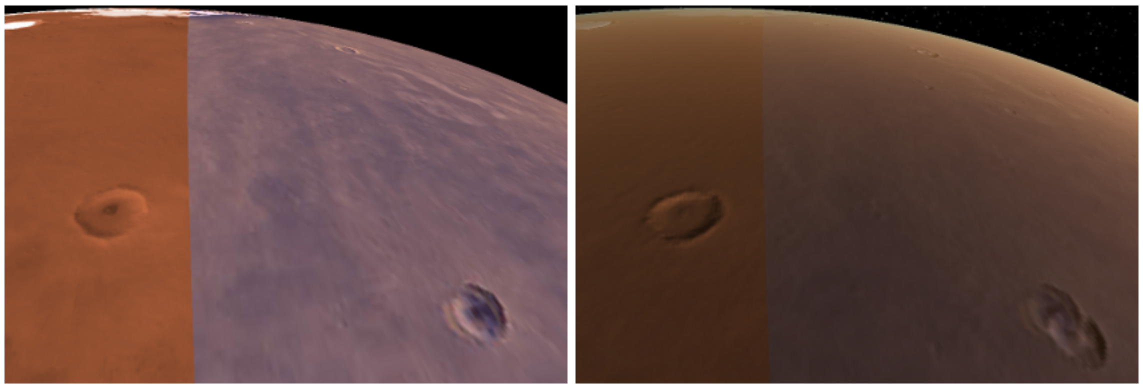


Figure 2.36: The MOC (left) and Viking (right) global color datasets.

the planet. The areoid also serves as a datum for topographic measurements, and is analagous to the Earth's geoid [32]. The maximum resolution offered by MOLA is 128 pixels/degree (0.463 km/pixel) for the entire planet and 512 pixels/degree (0.116 km/pixel) for either pole.

The High Resolution Imaging Science Experiment (HiRISE) [45] camera is a device on the Mars Reconnaissance Orbiter (MRO). HiRISE provides high-resolution imagery for focused areas of the planet in both equirectangular and polar stereographic projections. For the purpose of comparison, an image of Victoria Crater has a resolution of 237,099 pixels/degree (0.25 m/pixel) [46]. In addition, digital terrain models are generated by the HiRISE team from pairs of images taken of the same area.

Global color mosaics of Mars also exist, although they are typically not directly provided by NASA. A global mosaic of images created from NASA's Viking program is available through many channels [66]. Despite its apparent popularity, the dataset uses a planetographic coordinate system which differs from the planetocentric coordinate system used in recent missions [22]. The difference between the two is whether the calculation of latitude is performed with respect to an ellipsoid or a sphere [64]. As a consequence, the Viking imagery does not match MOLA or HiRISE. A global greyscale mosaic of images from the Mars Orbiter Camera (MOC) [40] on the MGS satellite is available [66]. It uses the more recent planetocentric coordinate system and matches MOLA and HiRISE. Figure 2.36 demonstrates the difference between these two global color datasets. The left image is of the planet without lighting; the red area on the left side is the MOC, and the brown area on the right is the Viking dataset. In the right image, lighting is turned on. Since the normals used for lighting are generated from MOLA, which MOC matches, there is no apparent error in MOC area of the image. However, Viking does not match MOLA, and therefore the craters appear twice in the Viking area of the image.

Chapter 3

Projective Grid Mapping for Planetary Terrain

GPU-based algorithms powerfully address the problems of planetary rendering. CPU-based algorithms, however, are not suitable; most focus on planar terrain, store a static mesh, and require lengthy preprocessing. In contrast, algorithms that take advantage of modern graphics hardware easily perform planetary rendering tasks such as data composition. For example, Kooima [33] stores the mesh in GPU memory, treating both height and color data as textures. Since each texel represents a scalar value, geographically overlapping texels are trivially combined. The GPU allows algorithms to manage the large amount of processing required for planetary rendering.

Consequently, we choose to modify projective grid mapping for planetary rendering. PGM provides a steady framerate and allows throttling based on the dimensions of the projective grid. Also, PGM can be implemented on any graphics hardware that supports vertex shaders, fragment shaders, and vertex texture lookups.

Our algorithm begins with a grid of points that spans the viewport (Figure 3.1). Each grid point corresponds to a ray that originates at the viewpoint. The grid points are projected along these rays onto a sphere, which serves as a reference surface for displacement. The result is a grid stretched across the sphere (Figure 3.2). Subsequently, each grid point is displaced by various height textures. During rasterization, the projected and displaced grid is interpreted as a mesh that approximates the visible terrain (Figure 3.3).

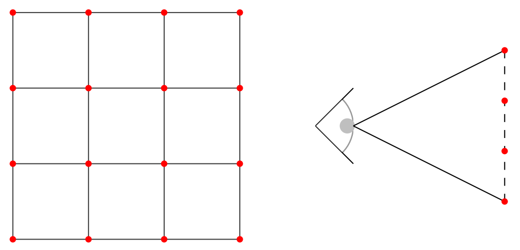


Figure 3.1: The projective grid and its arrangement over the viewport.

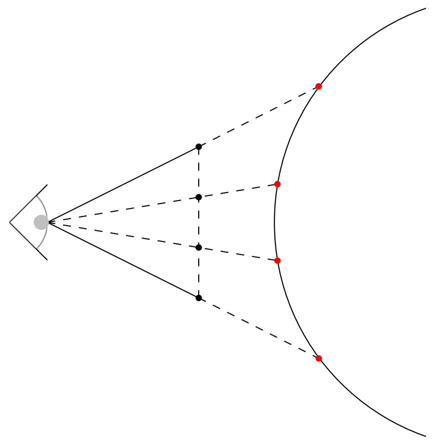


Figure 3.2: The rays and their sphere intersections.

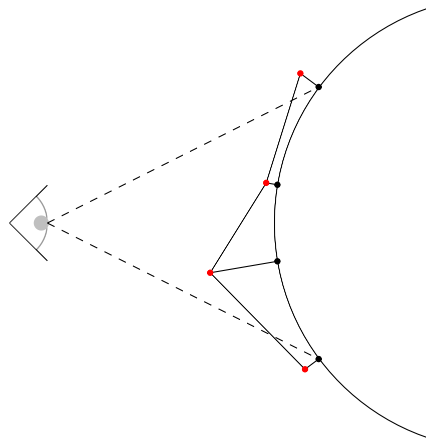


Figure 3.3: The displaced mesh that represents the terrain.

3.1 Algorithm Overview

Our algorithm is a combination of PGM and deferred texturing. As shown in Figure 3.4, the algorithm is divided into five steps, in which each step produces data buffers that are used in later steps.

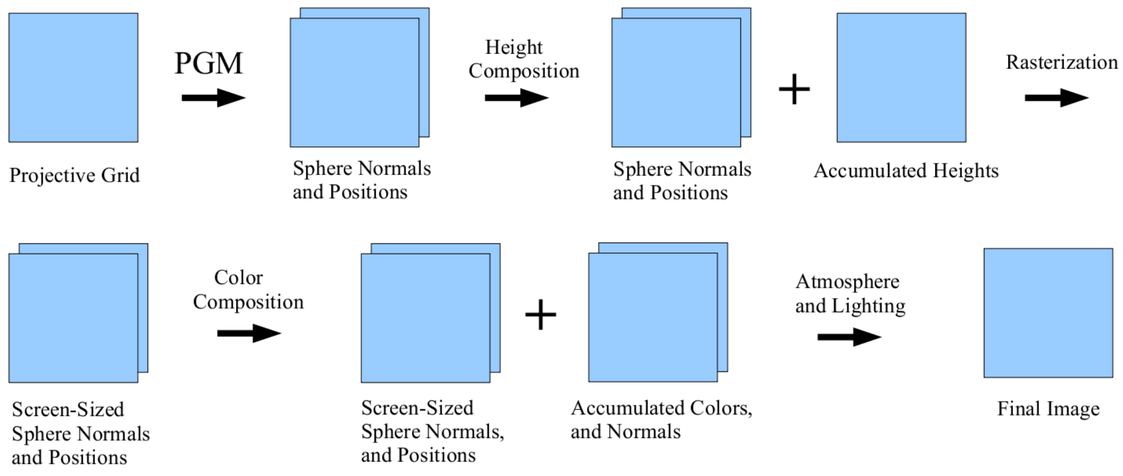

Projective Grid
PGM
Sphere Normals and Positions
Height Composition
Sphere Normals and Positions
+
Accumulated Heights
Rasterization
Screen-Sized Sphere Normals and Positions
Color Composition
Screen-Sized Sphere Normals, and Positions
+
Accumulated Colors, and Normals
Atmosphere and Lighting
Final Image

Figure 3.4: The steps and data flow of the algorithm.

The algorithm begins with PGM. Prior to projecting the grid, however, we must ensure that the planet will be adequately sampled. Two special entities, the reference sphere and the sampling camera, are used to achieve adequate sampling. When projecting the grid, a sphere is required as a reference for intersecting rays, for calculating geographic coordinates, and for displacing grid points. The ideal reference sphere closely approximates the terrain with respect to the average visible height and the geographic area of the planet that could affect the final image. In addition, the area beyond the horizon should be sampled to handle the case of a distant mountain that rises from behind the horizon. To meet these goals, two reference spheres are selected, and the results of ray intersection are interpolated. These are called the primary and secondary reference spheres.

In addition to the reference spheres, a camera that adequately samples the primary reference sphere is required. The ideal sampling camera produces all of the rays that intersect the area of the primary reference sphere that could be displaced into view. The sampling camera is based on the viewing camera and the visible area of the primary reference sphere. Selecting reference spheres and calculating the sampling camera are both view-dependent tasks, so they occur for every frame that the viewing camera changes.

With the reference spheres and the sampling camera, PGM proceeds by generating a ray for each grid point. Each ray is intersected with both reference spheres. As an optimization, ray-sphere intersection is reduced to the two-dimensional case of a ray and a circle, leading to simple and direct methods for selecting reference spheres and calculating the sampling camera. In addition, the two-dimensional intersection reduces the amount of shader code for ray intersection. The results are interpolated to achieve a gradual transition between detail at close range and detail from behind the horizon. Floating point precision issues are mitigated by generating positions relative to the viewpoint, which reduces the distance to the origin.

As shown in Figure 3.4, PGM outputs two buffers, which hold the positions and the sphere normals at those positions. These buffers constitute the projected grid and determine the area of the planet for which data is required. Using the projected grid, the next four steps accomplish deferred texturing, the goal of which is to sample and combine the height and color textures that affect this area.

The first of these steps is height composition, which accumulates height values for each height dataset over all of the projected grid points. To sample geographic data, this step uses the sphere normals, which correspond to geographic locations. The next step is rasterization, which uses the accumulated height buffer to displace the mesh and render it into screen-sized buffers. The color composition step uses the screen-sized position and sphere normal buffers to accumulate colors and surface normals, which are different from sphere normals in that they describe the surface of the terrain for lighting equations. In the last step, optimized versions of atmospheric equations by Sean O' Neil [51] and Phong lighting equations [1, 289-304] are applied using the accumulated color and normal buffers. The final buffer is drawn to the screen.

3.2 Two-Dimensional Reduction

Adequately sampling the planet with respect to a viewpoint and a reference sphere is a difficult three-dimensional problem; therefore, it is helpful to work in two dimensions. For this we refer to Fourquet *et al.* [19] who show that ray-terrain intersection with a spherical underlying surface can be reduced to two-dimensional cross sections consisting of a ray and a circle, as demonstrated in Figure 3.5. The cross section is formed by the plane containing the ray and a special ray called the *nadir*, the unit vector that points toward the center of the sphere. In the figure, the nadir is the ray that goes through the point CoP$_v$.

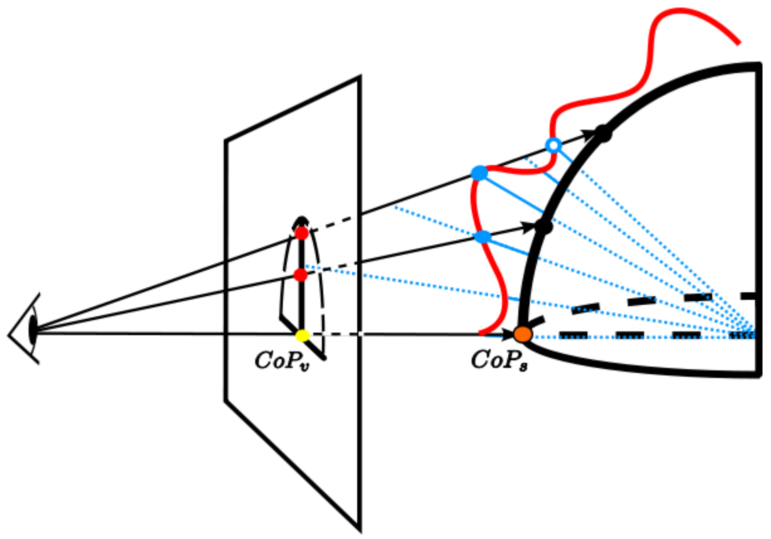


Figure 3.5: The two-dimensional reduction. [19]

Although Fourquet *et al.* [19] give equations to map points on the view plane to geographic coordinates, we elect to derive a trigonometry-based equation that results in less function calls when implemented on the GPU.

The goal of PGM is to calculate a ray-sphere intersection point and its corresponding sphere normal for each grid point. A typical approach would be to use a ray-sphere intersection algorithm to get the intersection point and then calculate the sphere normal at that point. However, we use the two-dimensional reduction to derive a trigonometric equation that directly calculates the sphere normal. Although the intersection point must eventually be calculated, the direct sphere normal equation is useful for selecting reference spheres and calculating the sampling camera. As described in Section 3.6, we use the sphere normal and the altitude to get the intersection point.

We derive the trigonometric equation that maps the elevation of an eye ray to the elevation of the sphere normal at the ray-sphere intersection point, as shown in Figure 3.6. The angle ω is the angle between the eye ray $\widehat{r}$ and the nadir $\widehat{n}$, and the angle γ is the angle between the sphere normal $\widehat{s}$ and the antinadir $\widehat{a}$, the opposite of the nadir. All vectors are assumed to be unit vectors unless otherwise noted.

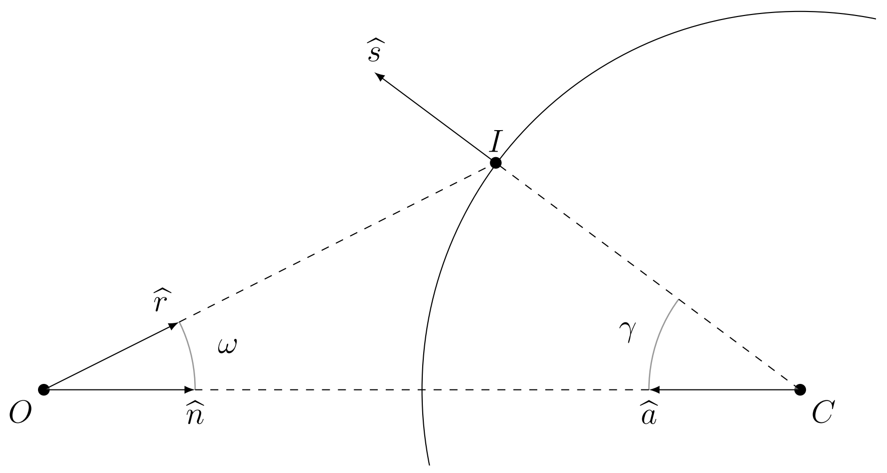


Figure 3.6: Rays and angles in the ray-circle intersection problem.

The concept of the two-dimensional reduction is important because it is used to select the reference spheres, calculate the sampling camera, and project the grid points onto the reference spheres. We work in radians since standard implementations of trigonometric functions use radians.

Given the ray angle ω, we will now show how to calculate the normal angle γ at the ray-circle intersection point. In Figure 3.7, the length r is the radius of the

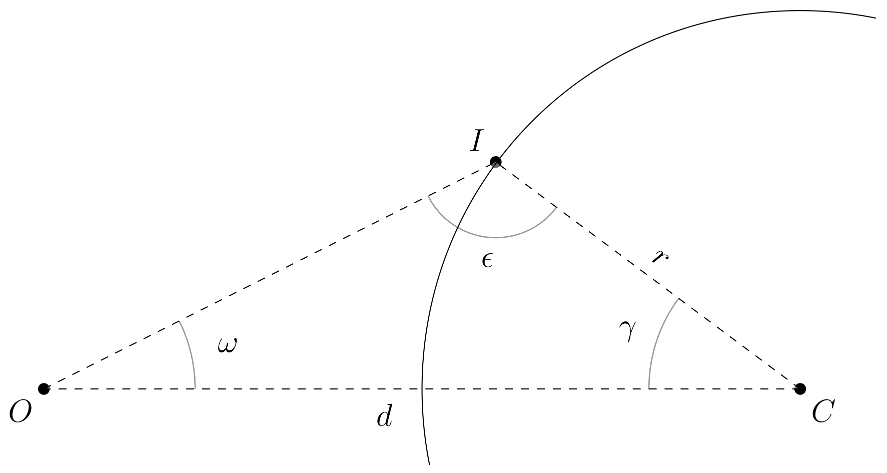


Figure 3.7: The ray-circle intersection problem.

circle, and d is the distance between the ray's origin O and the circle's center C. The intersection problem produces $\triangle OCI$ for which we know the length of two sides (d and r) and an angle (ω). Therefore, the law of sines can be used to obtain an equation for a second angle ϵ in the triangle, written as

$$\frac{r}{\sin\omega} = \frac{d}{\sin\epsilon}.$$

Solving this equation for ϵ results in

$$\epsilon = \text{asin}(\frac{d}{r}\ \sin\ \omega). \tag{3.1}$$

With two angles of $\triangle OCI$, the third angle is

$$\gamma = \pi - \omega - \epsilon,$$

which is the normal angle.

However this is, in fact, not the angle we set out to find. An ambiguity with respect to the law of sines leads to two possibilities, as shown in Figure 3.8. Unless the ray is tangent to the circle, it intersects the circle twice: once as it enters and again as it exits, corresponding to the front facing and the back facing intersection points respectively. The ambiguity results in two triangles $\triangle OCI_1$ and $\triangle OCI_2$. These triangles share two side lengths (r and d) and an angle (ω). Conversely, the triangles differ by two angles and a side, the first angles being ϵ_1 and ϵ_2 in each triangle and the second angles being γ_1 and γ_2 in each triangle.

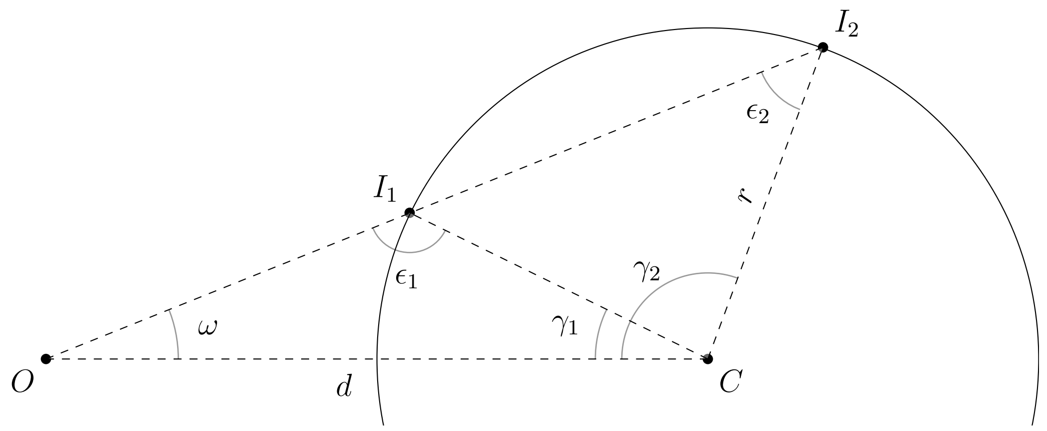


Figure 3.8: Ambiguous solutions using the law of sines.

This ambiguity is resolved using the observation that ϵ_2 in $\triangle OCI_2$ is acute and ϵ_1 in $\triangle OCI_1$ is obtuse and their sum is π [59]. The acute angle ϵ_2 is a direct result of Equation 3.1 given previously. The obtuse angle ϵ_1 and is found using

$$\epsilon_1 = \pi - \epsilon_2. \tag{3.2}$$

The calculation of the normal angles γ_1 and γ_2 for both triangles remains the same:

$$\gamma_1 = \pi - \omega - \epsilon_1,$$

$$\gamma_2 = \pi - \omega - \epsilon_2.$$

Finally, we use substitution to find equations for the normal angle as a function of the ray angle. The equation for γ_1 can be simplified by substituting ϵ_1 for Equation 3.2 to get

$$\gamma_1 = \pi - \omega - (\pi - \epsilon_2),$$

which reduces to

$$\gamma_1 = \epsilon_2 - \omega.$$

Substituting ϵ_2 with Equation 3.1, we obtain the direct equation for the front facing intersection. Similarly, we substitute Equation 3.1 into $\gamma_2 = \pi - \omega - \epsilon_2$ to obtain the direct equation for the back facing intersection. Together, the front facing and back facing equations are:

$$\gamma_1 = \text{asin}(\frac{d}{r} \sin \omega) - \omega, \tag{3.3}$$

$$\gamma_2 = -\text{asin}(\frac{d}{r} \sin \omega) - \omega + \pi. \tag{3.4}$$

3.3 Sampling Issues

Using the two-dimensional reduction, we now address three obstacles that affect sampling. To achieve adequate sampling, grid points should not be excessively displaced toward the viewer and should never be displaced away from the viewer. In addition, only the area of the planet that could possibly affect the final image should be sampled.

3.3.1 Stretching and Shrinking

The first obstacle is stretching, which occurs when grid points are excessively displaced toward the viewer and away from the center of the planet. The greater the displacement distance, the larger the triangles of the mesh appear in screen space. Another consequence is that parts of the mesh fall outside of the view frustum.

The effects of shrinking are apparent in the extreme case where the mesh is maximally displaced. The minimum and maximum radii of the planet, known *a priori*, define all of the possible positions of a vertex, as shown in Figure 3.9. Consider the extreme case where the reference sphere is equal to the minimum sphere and the entire mesh is displaced to the height of the maximum sphere, as demonstrated in Figure 3.10. The arc length of the non-displaced mesh M_1 is smaller than the arc length of the maximally displaced mesh M_2.

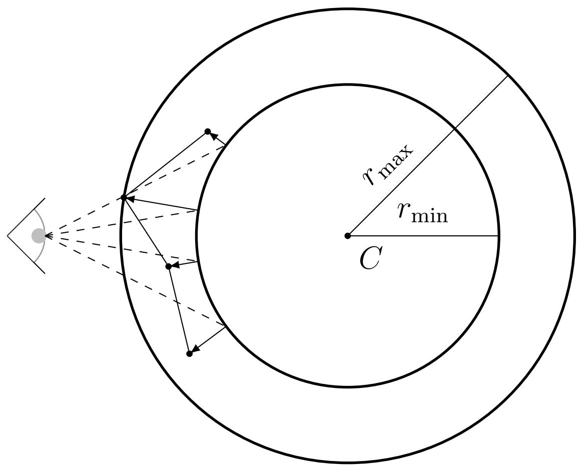


Figure 3.9: The minimum and maximum spheres.

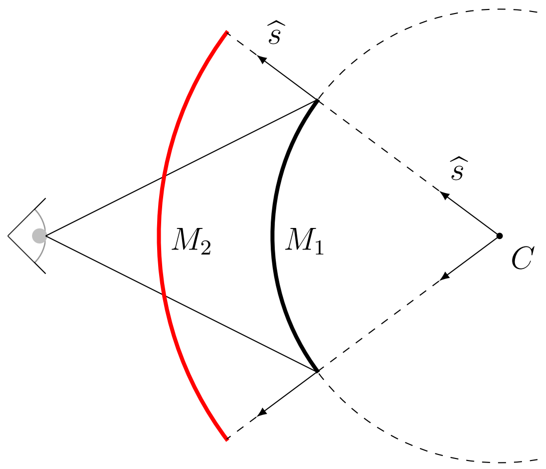


Figure 3.10: The effects of stretching.

Additionally, Figure 3.11 demonstrates the effects of stretching within the visualization. In the left image, the mesh is shown before height displacement, and the viewing frustum, marked with red lines, approximately fits the mesh. In the right image, however, the mesh is shown after height displacement. Because the left side of the mesh is displaced toward the viewer, some edges of the mesh extend beyond the viewing frustum. Since our objective is to generate approximately pixel-sized triangles that do not fall outside of the viewing frustum, the displacement distance must be minimized by choosing an appropriate reference sphere.

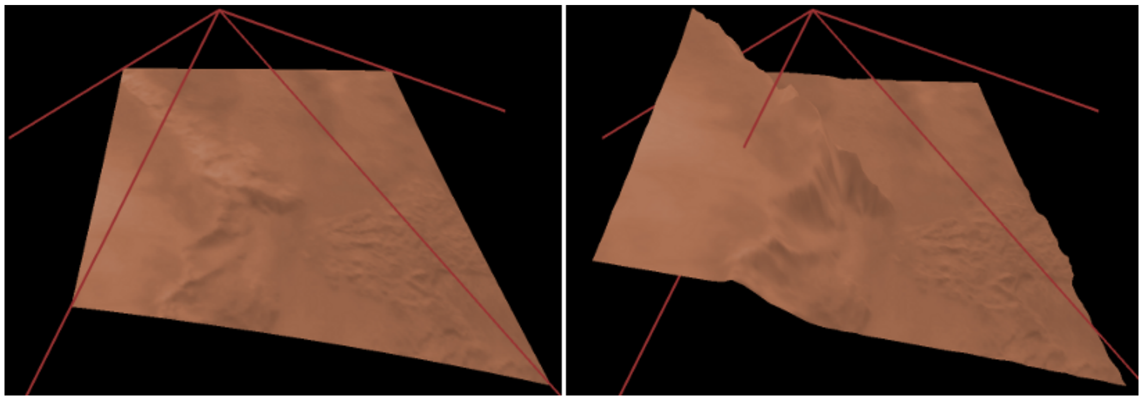


Figure 3.11: Stretching and shrinking within the visualization.

The second obstacle is shrinking, which occurs when grid points are displaced away from the viewer and toward the center of the planet. The effect of shrinking is that the edges of the mesh fall within the view frustum and are visible to the viewer. This problem is illustrated in Figure 3.12. Also, Figure 3.11 demonstrates shrinking within the visualization. In the right image, the right side of the mesh is displaced away from the viewer and into the viewing frustum. Although shrinking can be solved by using the minimum sphere as the reference sphere, this results in a large amount of stretching. A method of choosing a reference sphere that solves both of these problems must be devised.

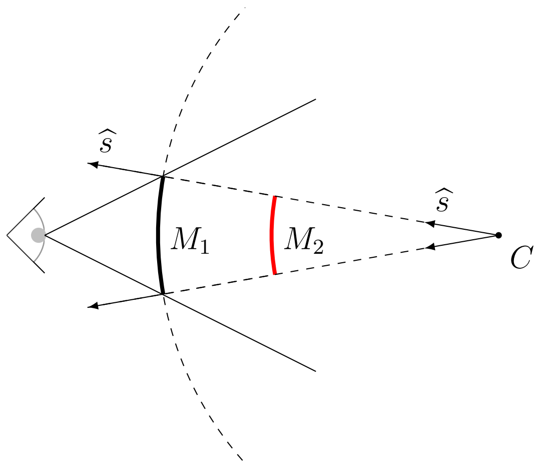


Figure 3.12: The effects of shrinking.

3.3.2 Pertinent Area

The *pertinent area* is the geographic area of the planet that affects the final image. The third obstacle comprises undersampling and oversampling this area. The eye rays that intersect the pertinent area on the sphere are referred to as *pertinent rays*.

The pertinent area is divided into three parts, as shown in Figure 3.13. The frustum is represented by the solid line segments. The tangent ray $\widehat{t}$ separates the group of eye rays that intersect the planet and the group of eye rays that do not. The visible area, marked in green, is contained within the view frustum. The area marked in red is the occluded area; it is the part of the sphere that could rise from behind the horizon. The occluded area is determined by the maximum radius.

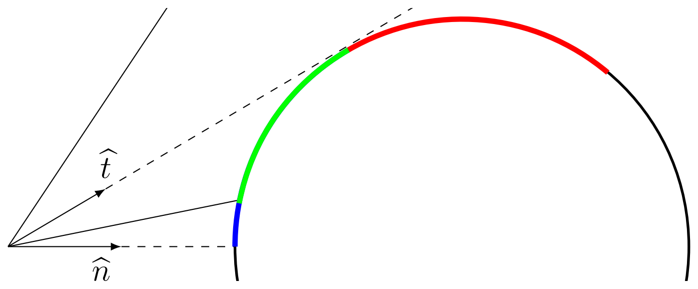


Figure 3.13: The pertinent area of the sphere relative to the viewpoint.

Using the appropriate reference sphere and sampling the entire pertinent area allows mountains within the occluded area to be seen in the visualization, as shown in Figure 3.14. In the top image, the mesh is partially projected onto the geographic area of the mountain; as a result, the far edge of the mesh is visible when the mesh is displaced. In the bottom image, however, the mesh is projected over enough of the mountain's geographic area so that the far edge is not visible when the mesh is displaced. Figure 3.15 illustrates these situations from a third-person perspective.

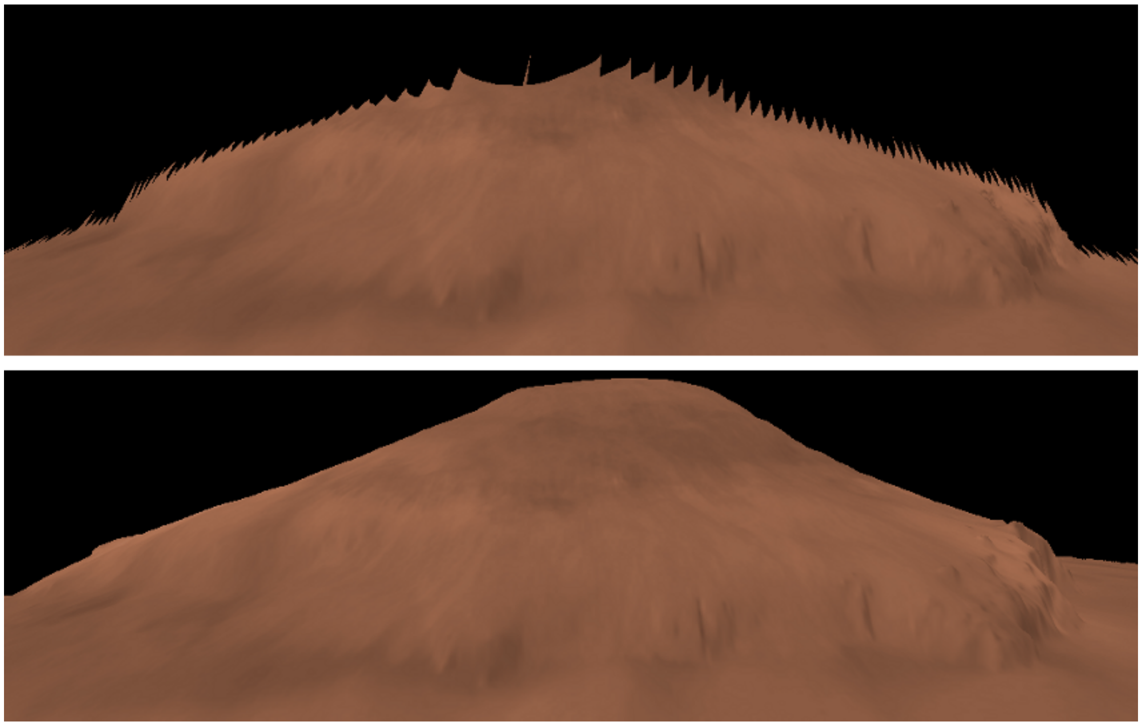


Figure 3.14: The occluded area should be sampled.

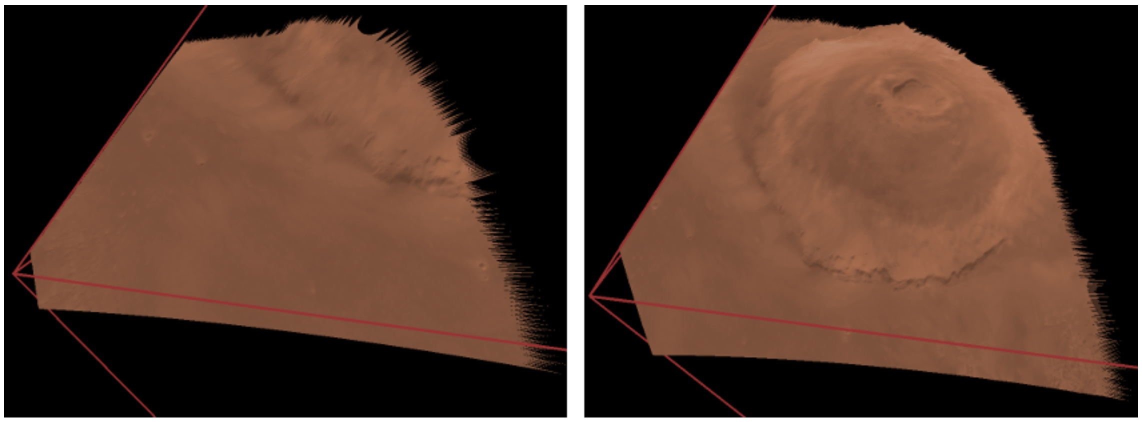


Figure 3.15: The occluded area problem from a third-person perspective.

Similarly to the occluded area, the blue region below the view frustum in Figure 3.13 is not visible; however, it could also be displaced into view, as Figure 3.16 illustrates. This area begins at the point that the nadir $\widehat{n}$ intersects the sphere and ends where the bottom of the frustum intersects the sphere.

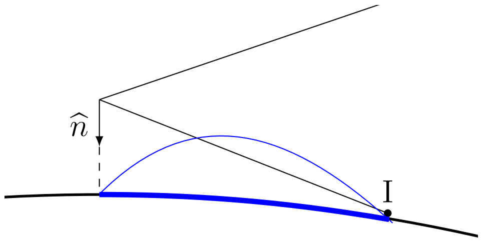


Figure 3.16: The area below the frustum could affect the final image.

Figure 3.17 demonstrates this situation within the visualization. The viewing camera is marked in blue, and the ideal camera that samples the area below the viewing camera is marked in red. In the left image, the projected mesh is not displaced, and it is clear that the viewing camera does not sample the mountain directly below. In the right image, the mesh is displaced and the mountain below the viewing camera rises into view. Although Livny *et al.* [37] use a plane as an underlying surface, their discussion on sampling the area below the frustum is similar.

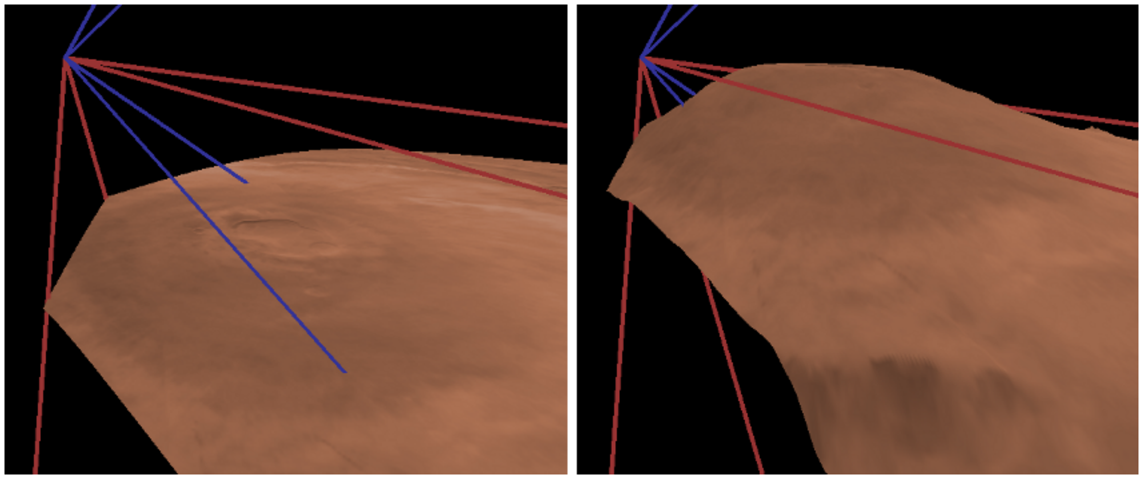


Figure 3.17: A mountain affecting the view frustum in the visualization.

In preparing for PGM, the reference spheres ensure that no stretching or shrinking occurs and that the pertinent area is visible, while the sampling camera ensures that all of the pertinent rays are generated.

3.4 Reference Spheres

Two reference spheres are selected. The primary reference sphere minimizes stretching and avoids shrinking. The secondary reference sphere, when ray cast, produces intersection points that represent the occluded area. The PGM step projects the grid points on both spheres and interpolates the results. A reference sphere is defined by a center and a radius. Because the center of the planet is the same for both spheres, we are concerned only with the radii. Calculation of the radii occurs on the CPU before PGM.

3.4.1 Primary Reference Sphere

We give two methods for selecting the radius of the primary reference sphere. The first method uses a global raster that resides in core memory. This raster contains the distance from the center of the planet, called the planetary radius, over the entire surface of the planet. At the beginning of the program, the raster is loaded. During rendering, the geographic location (λ, ϕ) of the viewer is calculated from the antinadir $\widehat{a}$, which always points from the center of the planet toward the eye.

$$\lambda = \operatorname{atan2}(\widehat{a}_x, \widehat{a}_z)$$
$$\phi = \operatorname{asin}(\widehat{a}_y)$$

This geographic location is used to query the global raster for the planetary radius of the terrain at the viewer's position. This value suffices as an estimate and is efficient to compute. However, if the global raster is of low resolution and detailed height maps are active, this value is inexact relative to the high-resolution data and may lead to shrinking. In addition, this method is not suitable for extreme cases where the planetary radius at the viewpoint is much different from the terrain being rendered, such as when the user is looking down from a cliff.

The second, more precise method is to use the minimum visible planetary radius from the previous frame. During the rasterization step (Section 3.7.2), the planetary radius for each fragment is written to a screen-sized buffer. To calculate the minimum value in the buffer, a min mipmap is constructed on the GPU. A min mipmap is similar to an average mipmap, except that a texel represents the minimum of the four texels in the previous mipmap level. After the min mipmap is constructed, the texel in the highest level of the mipmap is read from the GPU to the CPU and saved for the next frame.

The value obtained for the primary sphere radius results in a more robust rendering that minimizes stretching while avoiding shrinking. In addition, in the future it may be possible to use the min mipmap for collision detection with the composited terrain.

Figure 3.18 demonstrates the difference between the two methods. The user is positioned at the top of a cliff looking into a deep valley. The first image shows the

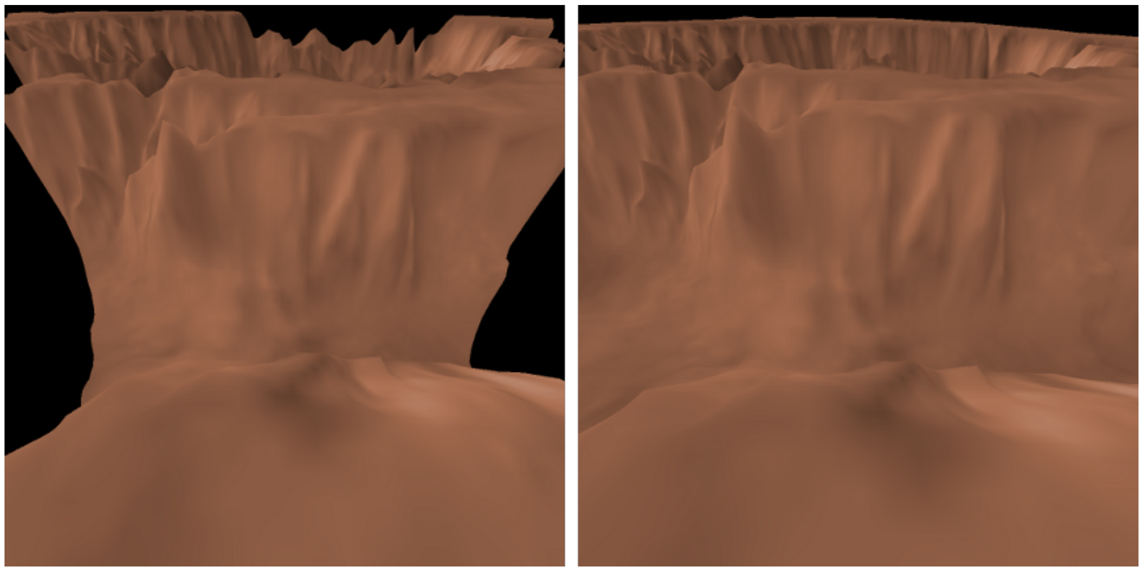


Figure 3.18: The two methods for the primary reference sphere.

result of using the local radius, which is the radius of the cliff, and the second image shows the result of using the minimum visible radius. In the first image, the edges of the mesh are visible on the sides, while in the second image the edges are not.

3.4.2 Secondary Reference Sphere

The goal of the secondary reference sphere is to capture the occluded area of the primary reference sphere. Although the projected grid points could be stretched beyond the horizon of the primary reference sphere, such an operation would ruin the view-dependent nature of ray casting and require additional shader code. A better method is to find a second reference sphere that allows the occluded area to be ray cast.

To calculate the radius of the secondary reference sphere, the extent of the occluded area is needed. This extent is defined by the maximum possible normal angle. This upper bound is dependent on the viewpoint, the primary reference sphere, and the maximum sphere. To find the upper bound, the tangent ray of the primary reference sphere is intersected with the maximum sphere. The upper bound is then the normal angle for the back facing intersection point, as shown in Figure 3.19. Let the angle τ be the angle of the tangent ray $\widehat{t}$ to the primary reference sphere **P**. The ray intersects the maximum sphere **M**, and the vector $\widehat{s}$ represents the sphere normal at

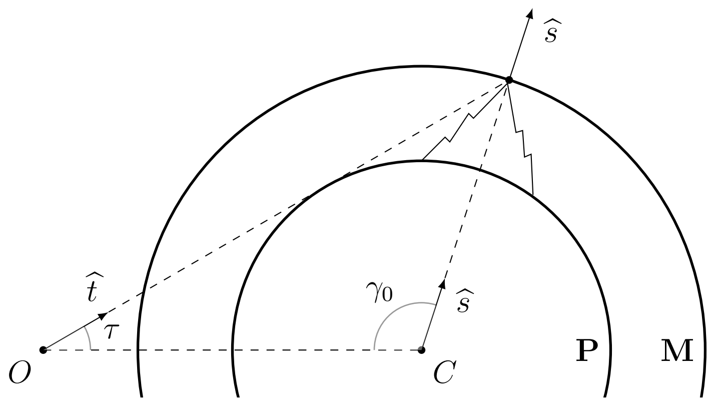


Figure 3.19: The upper bound of the normal angle.

the back facing intersection point. The normal angle for $\widehat{s}$ is γ_0. Consider an eye ray that is just above $\widehat{t}$, that is, a ray with a ray angle greater than τ. Because the maximum sphere limits the distance of displacement, no point on the sphere could be displaced so as to affect this ray.

To calculate the upper bound of the normal angle, the angle of the tangent ray to the primary reference sphere is used, as shown in Figure 3.20. The tangent ray $\widehat{t}$ forms a right angle to the line segment $\overline{CI}$. The right triangle implies that $\sin \tau = r/d$, which can be rewritten as

$$\tau = \operatorname{asin}(r/d). \tag{3.5}$$

To calculate the upper bound of the normal angle γ_0, we use the tangent angle τ and Equation 3.4 to calculate the normal angle at the back facing intersection of the tangent ray with the maximum sphere.

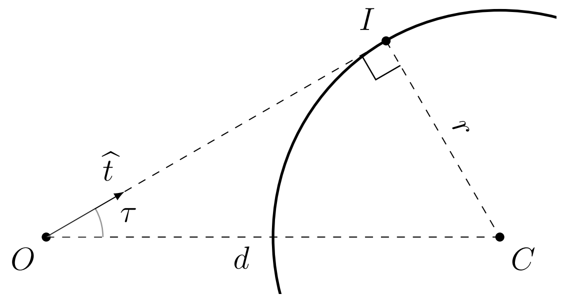


Figure 3.20: The tangent angle.

With the upper bound we will now show that, in most cases, there exists a smaller sphere that allows the occluded area of the primary reference sphere to be ray cast, and we will give a method for calculating the radius of the smaller sphere. As Figure 3.21 shows, the ray $\widehat{t_p}$ is tangent to the primary reference sphere $\mathbf{P}$ and intersects the maximum sphere $\mathbf{M}$ as it exits the sphere. The back facing intersection produces the normal angle γ_0 that represents an upper bound on all normal angles for this viewpoint. The secondary reference sphere $\mathbf{S}$ should be the sphere with a tangent

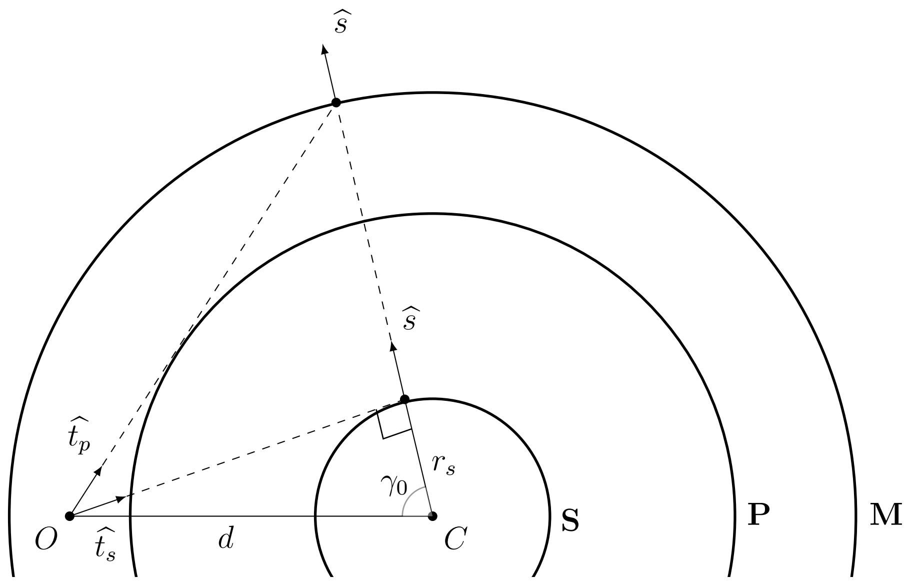


Figure 3.21: The radius of the secondary reference sphere.

ray $\widehat{t_s}$ that results in a normal angle of γ_0 when intersected with **S**. The distance d to the center of the planet is the same for all spheres, and the ray $\widehat{t_s}$ forms a right triangle with respect to **S**. Therefore, the radius r_s of the secondary reference sphere is included in the equation $\cos\gamma_0 = r_s/d$, which can be written as

$$r_s = d \cos \gamma_0. \tag{3.6}$$

When the tangent ray $\widehat{t_s}$ is intersected with **S**, it produces the normal angle that represents the farthest possible point on **P** that could rise into view.

Figure 3.22 demonstrates how the occluded area of the primary reference sphere becomes part of the visible area on the secondary reference sphere. The red area on the primary reference sphere is a subset of the visible geographic area on the secondary reference sphere.

The secondary reference no longer exists when Equation 3.6 returns a negative value for the radius. Because the ratio between the minimum sphere and the maximum sphere for actual planets is small, the distance at which this occurs is sufficiently far away to replace the planet with an impostor.

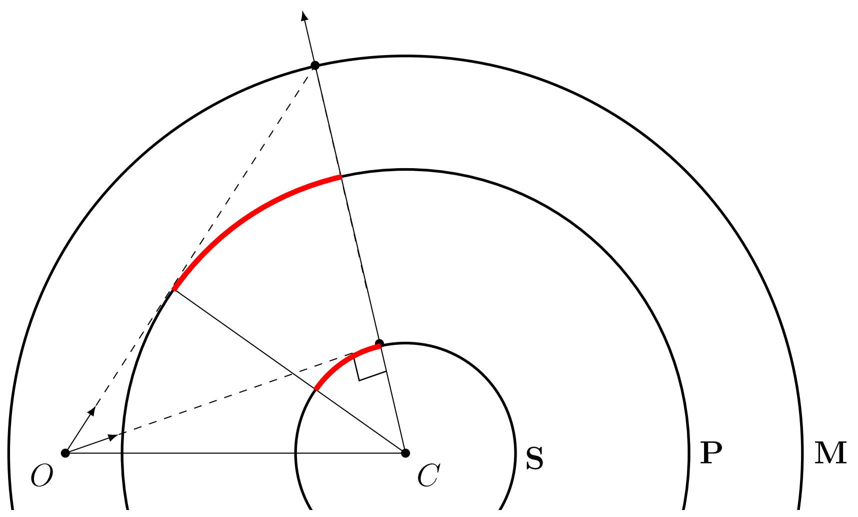


Figure 3.22: The occluded area is visible on the secondary reference sphere.

3.5 Sampling Camera

The viewing camera consists of a position, an orientation, and a frustum. The position of the viewing camera is simply the viewpoint. The orientation is a coordinate system that defines coordinates relative to the camera. This coordinate system is typically called eye space. The three basis vectors that make up this coordinate system form a matrix, called the view matrix, which transforms points and vectors from world space to eye space. Finally, the viewing camera's frustum contains all of the visible objects in the scene; a special matrix called the projection matrix transforms points into the canonical view volume, where visible objects exist within the cube defined by $(-1, -1, -1)$ and $(1, 1, 1)$.

The sampling camera also consists of a position, an orientation, and a frustum. Although the sampling camera and the viewing camera are both positioned at the viewpoint, the sampling camera has a different orientation and frustum. The orientation defines the coordinate system relative to the sampling camera, called sampling camera space. The matrix that transforms points and vectors from world space to sampling camera space, called the sampling camera matrix, is constructed from this coordinate system. Finally, the sampling camera's frustum contains the parts of the sphere that should be sampled. The attributes of the frustum are defined with respect to the near plane in sampling camera space, as Figure 3.23 demonstrates. These attributes are left, right, bottom, top, near, and far.

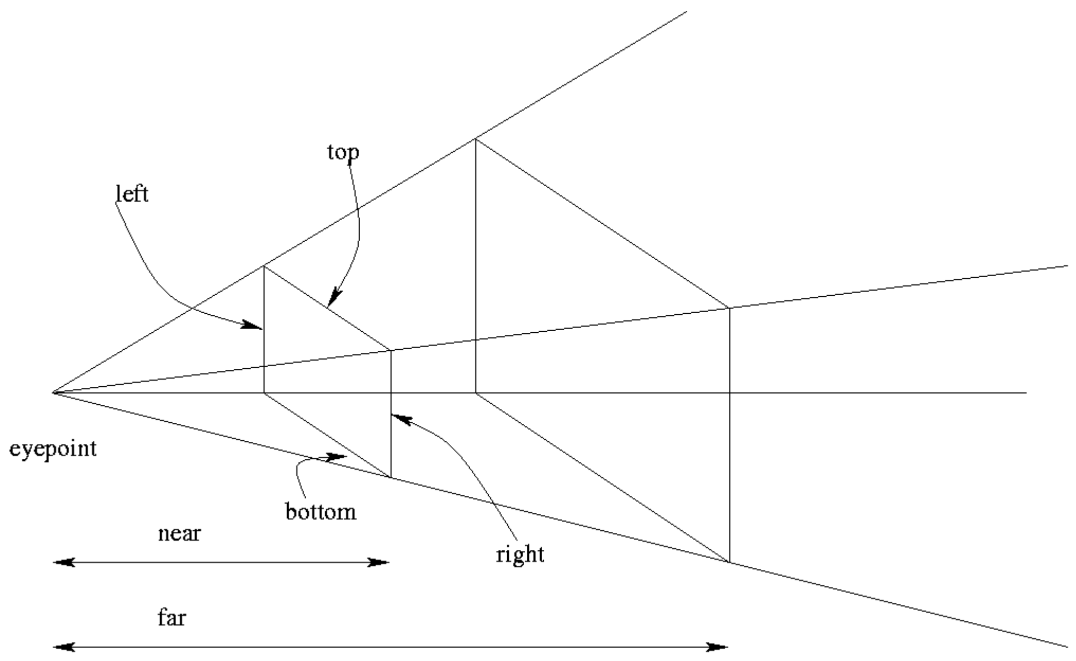


Figure 3.23: The attributes of the viewing frustum. [67]

Sampling camera generation is concerned with two goals. The first is to find the three basis vectors that define the sampling camera's orientation; these vectors are used to construct the sampling camera matrix. The second goal is to find the frustum attributes, which are used to calculate a projection matrix for the sampling camera's frustum. The inverse of the projection matrix is used in PGM to convert a grid point to its corresponding eye ray.

The ideal sampling camera generates all possible pertinent rays for a given viewpoint and reference sphere. The set of all pertinent rays is the intersection of two sets of rays. The first set consists of all rays generated by the view frustum, as shown in Figure 3.24. In addition, the first set also contains the group of all rays between the nadir and the bottom of the view frustum, as shown in Figure 3.25.

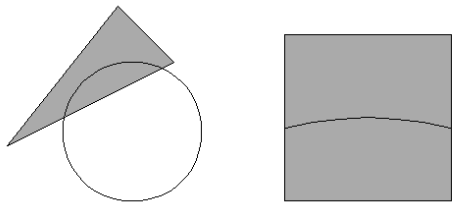


Figure 3.24: All of the eye rays for a given viewing camera.

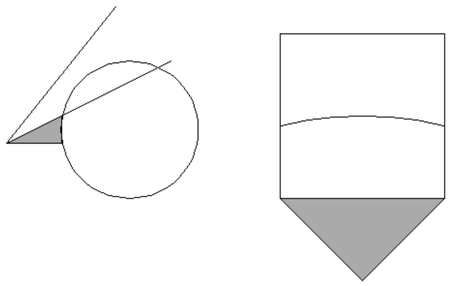


Figure 3.25: The pertinent rays below the frustum.

The second set of rays contains all possible rays originating from the viewpoint that intersect the reference sphere. These rays define an infinite cone, as shown in Figure 3.26. The intersection of these two sets is illustrated in Figure 3.27. In practice, it is difficult to intersect a cone with a frustum. Instead, we approximate the cone of all possible eye rays with a frustum that entirely contains the cone, as shown in Figure 3.28. This frustum is known as the visible sphere frustum since it includes the entire visible area of the reference sphere.

Our algorithm finds the intersection between the visible sphere frustum and the view frustum. If this frustum does not already contain the nadir, the frustum is extended to include the rays between the nadir and the bottom of the view frustum. If this frustum already contains the nadir, then no extension is made.

We observe that these two frusta share an apex, namely the viewpoint. This greatly simplifies finding their intersection. With the set of intersection rays, we generate both the view matrix and the projection matrix. The view matrix is generated by choosing a forward direction based on the intersection rays. The projection matrix

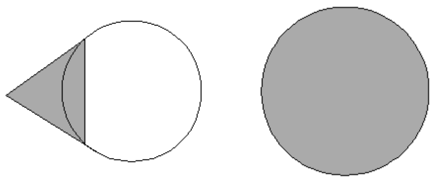


Figure 3.26: The cone of rays that contains the visible part of the planet.

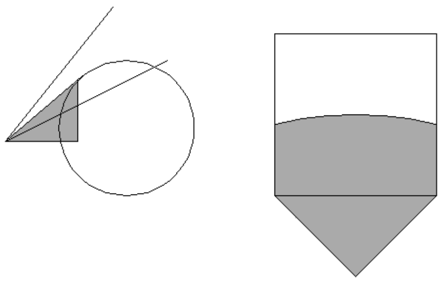


Figure 3.27: The intersection of the sets of rays, which contains all pertinent rays.

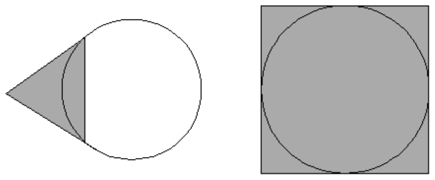


Figure 3.28: The cone of rays approximated with a frustum.

is generated by intersecting the intersection rays with the near plane, defined by the near distance and sampling camera space. The minimum and maximum values for x and y of the intersections are found and used as a parameters to build a frustum.

Figure 3.29 illustrates how the sampling camera should operate within the visualization. In the left image, the viewer is looking along the nadir. The frustum of the sampling camera, marked in red, is aligned with the sides of the screen. The right image shows this situation from a third-person perspective. The edges of the mesh appear jagged where the rays no longer hit the secondary reference sphere; however, these jagged edges are not visible from the viewer's perspective because the mesh ends within the occluded area. Additionally, Figure 3.30 shows the case where the viewing camera is pointed up and right from the nadir. Here, the sampling camera should be oriented so that the top of the frustum is tangent to the horizon in the direction of view. This rotation minimizes the number of eye rays that miss the reference sphere when near the surface. Although no difference is made when viewing the entire planet (left image), many rays are excluded at the surface (right image).

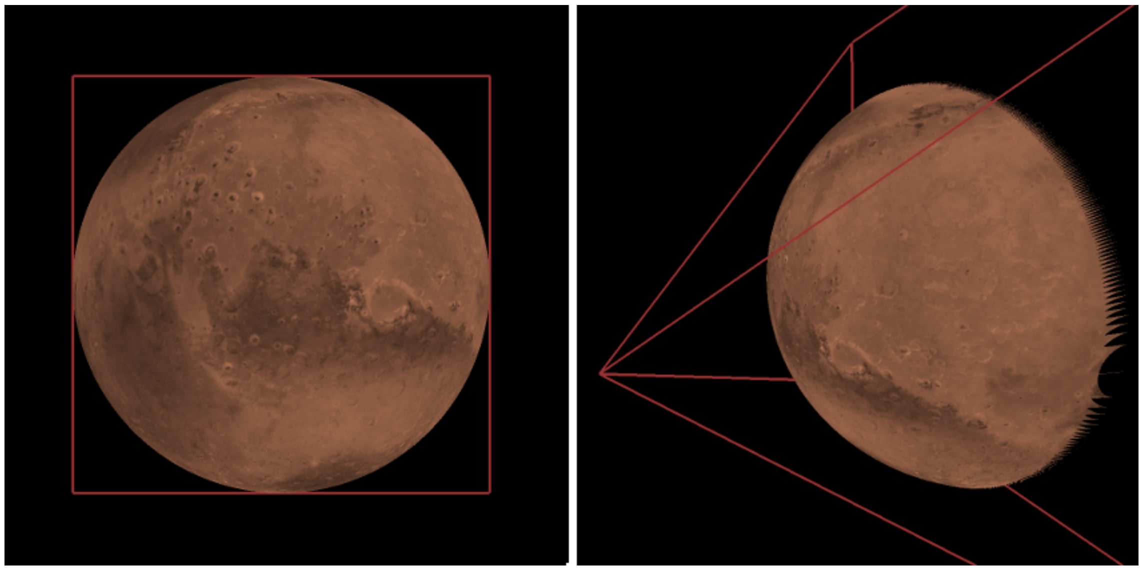


Figure 3.29: The sampling camera within the visualization.

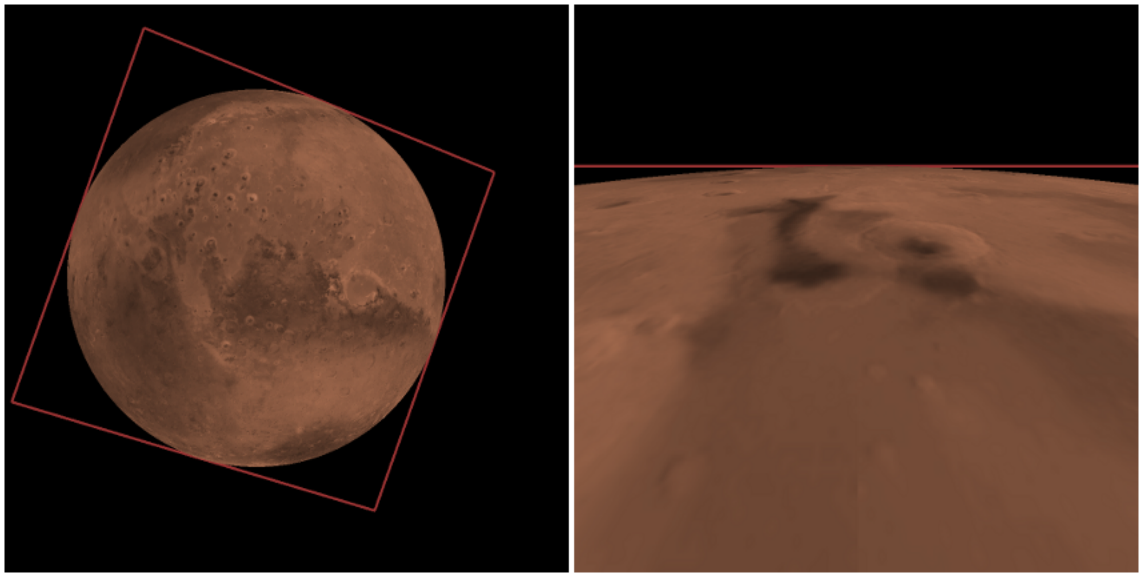


Figure 3.30: The sampling camera should fit the horizon.

3.5.1 Corner Rays

Frustum-frustum intersection requires that each frustum be defined by the rays that form its corners. There are four corner rays per frustum, and each ray has the viewpoint as its origin. Therefore, the first step of the sampling camera algorithm is to calculate the corner rays of the view frustum and the visible sphere frustum.

The corner rays of the view frustum are obtained by calculating their coordinates on the near plane, which is chosen in advance. The near plane is parallel to the xy-plane and is defined by the distance n along the negative z-axis. As shown in Figure 3.31, the corners of the view frustum are rays that intersect the near plane at the coordinates $(l, b, -n)$, $(r, b, -n)$, $(r, t, -n)$, and $(l, t, -n)$ in eye space, where l, r, b, t, n, and f correspond to left, right, bottom, top, near, and far, respectively. These components are chosen by the application programmer, but they can also be obtained from the projection matrix of the viewing camera using special equations.

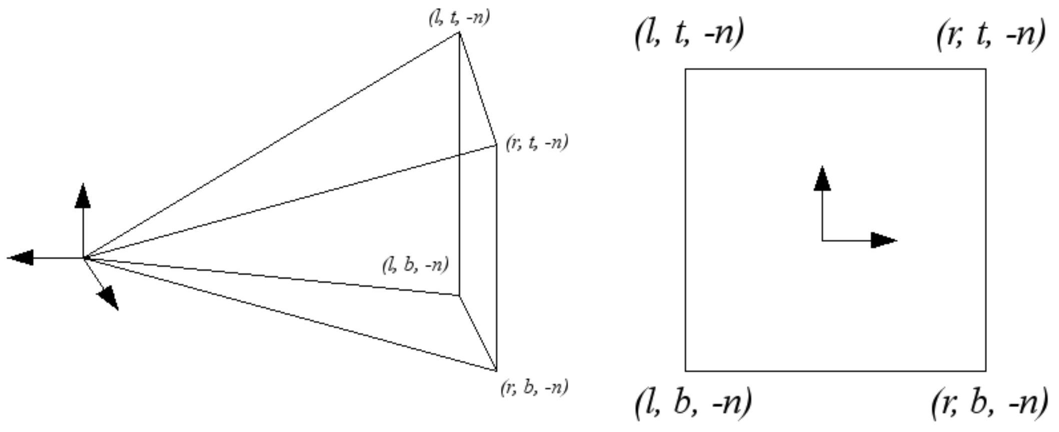


Figure 3.31: The corner rays of the frustum intersect the near plane.

The visible sphere frustum is calculated as follows. This frustum has a different orientation than that of the view frustum; therefore a coordinate system must be created that describes the orientation. Later, a transformation matrix derived from this coordinate system will be used to transform the visible sphere frustum's corner rays into the same coordinate system as the view frustum's corner rays.

We ensure that the visible sphere frustum is always oriented directly along the nadir, so the central axis of the frustum is the same as the nadir. However, the visible sphere frustum can have any rotation about the nadir (Figure 3.32). The view frustum, drawn in red, represents the viewport and the final image.

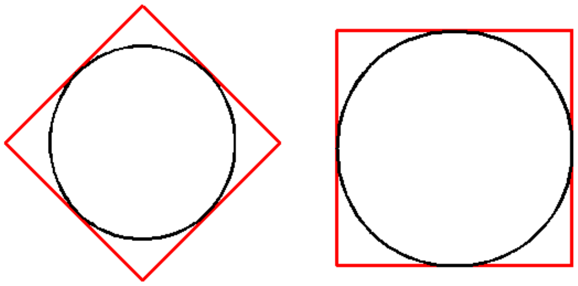


Figure 3.32: The visible sphere frustum can be rotated along its central axis.

In choosing this rotation, we consider the case in which the entire planet can be seen by the viewer (Figure 3.33) and the case in which the viewer is standing on the surface of the planet (Figure 3.34). The view frustum is marked in green. In the first case, the rotation does not matter because the visible sphere frustum is entirely within the view frustum; the intersection of the two frusta cannot exclude any eye rays that miss the sphere. Yet in the second case, the rotation should be chosen so as to minimize the number of rays in the visible sphere frustum that do not intersect the sphere. Although a rotation that is not aligned with the view frustum results in many rays that miss the sphere, an aligned visible sphere frustum minimizes the number of rays that miss the sphere.

A rotation aligned with the view frustum is obtained by using the viewing camera's forward vector along with the nadir to form the coordinate system of the visible sphere frustum. Using the viewing camera's forward vector $\widehat{f}$ and the nadir $\widehat{n}$, the basis vectors are

$\widehat{b} = -\widehat{n}$,

$\widehat{r} = \| \widehat{n} \times \widehat{f} \|$, and

$\widehat{u} = \| \widehat{r} \times \widehat{n} \|$,

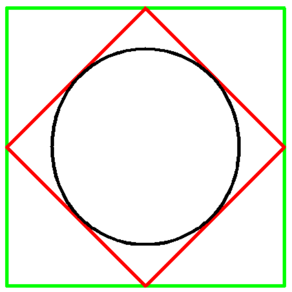


Figure 3.33: The view frustum may contain the entire planet.

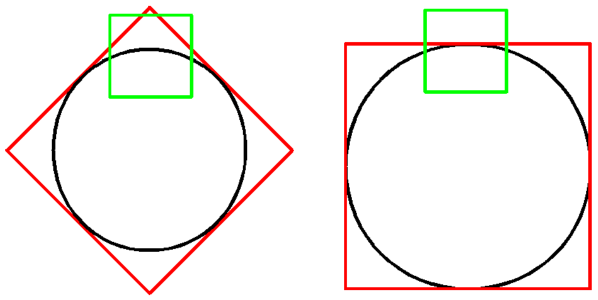


Figure 3.34: The view frustum may contain part of the planet.

where the backward vector $\widehat{b}$, right vector $\widehat{r}$, and up vector $\widehat{u}$ correspond to the z-, x-, and y-axes. We assume a right-handed coordinate system with the camera facing down the negative z-axis by default.

Now that we have the coordinate system relative to the visible sphere frustum, we need to find the corner rays of the visible sphere frustum. We now present a method for finding the corner rays relative to this coordinate system.

The visible sphere frustum is symmetrical about all three axes. As such, the corner rays form a square on the near plane relative to the coordinate system of the visible sphere frustum. To get the corner rays' coordinates on the near plane, half of the length s of the square's side is needed. In order to find this value, note that each of the four sides of the frustum correspond to a plane that is tangent to the reference sphere. Within each plane there exists an eye ray that is tangent to the sphere. We intersect this tangent ray with the near plane to find its distance from the center of projection on the near plane, *i.e.* the intersection of the nadir and the near plane. This distance corresponds to $h = s/2$.

The distance h is calculated using the angle τ made by a tangent ray. Figure 3.35 demonstrates that a right triangle is formed by the tangent ray and the line segment from the center to the tangent point. The tangent angle τ is found using Equation 3.5. A right triangle is formed with the near plane, implying that $\tan \tau = \frac{h}{n}$. Therefore, the equation

$$h = n \operatorname{atan} \tau$$

can be used to calculate the tangent distance h from the z-axis. The four corner rays are $(-h, -h, -n)$, $(h, -h, n)$, (h, h, n), and $(-h, h, n)$ in coordinates relative to the visible sphere frustum.

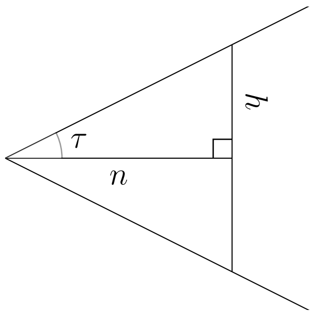


Figure 3.35: The height of the visible sphere frustum.

3.5.2 Corner Ray Transformation

The next step transforms the corner rays for both frusta into the same coordinate system. This is necessary for frustum-frustum intersection.

We choose to intersect the frusta in a coordinate system called rotated eye space. The best rotation for the viewing camera is one that is aligned with the horizon; this can be achieved by rotating the viewing camera so that the yz-plane includes the center of the planet, as shown in Figure 3.36. Rotated eye space is the same as eye space, except that it is rotated so that the yz-plane includes the center of the sphere.

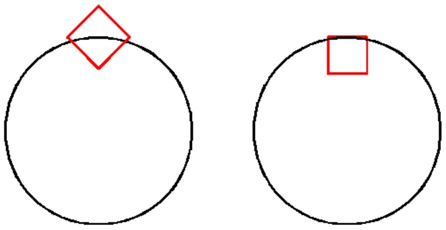


Figure 3.36: The view frustum should fit the horizon.

The purpose of this rotation is to ensure that the top edge of the sampling camera's frustum is parallel to the horizon, minimizing the number of rays cast that do not intersect the sphere.

First, the corners of the view frustum are transformed into rotated eye space. This coordinate system is found by building a set of three basis vectors from the forward direction of the camera $\widehat{f}$ and the nadir $\widehat{n}$, both in eye space. This is similar to the coordinate system of the visible sphere frustum, except that rotated eye space retains the forward direction of the viewing camera. The three basis vectors of the rotated eye frame are $\widehat{b}$, $\widehat{r}$, and $\widehat{u}$ and are calculated as follows:

$$\widehat{b} = -\widehat{f}$$

$$\widehat{r} = \|\widehat{b} \times \widehat{n}\|$$

$$\widehat{u} = \|\widehat{b} \times \widehat{r}\|$$

These vectors are used to build a transformation matrix that transforms points and vectors from eye space to rotated eye space. Each of the view frustum corner rays are multiplied by this matrix to obtain their equivalent in rotated eye space.

Next, the corner rays of the visible sphere frustum must be transformed into eye space before being transformed into rotated eye space. A transformation matrix is constructed that takes vectors from the coordinate system relative to the visible sphere frustum to world space. This matrix is built using the three basis vectors for the visible sphere frustum found previously. We then use the view matrix and the rotated eye matrix to transform the corner rays into rotated eye space.

3.5.3 Frustum-Frustum Intersection

With the corner rays for each frustum represented in the same coordinate system, frustum-frustum intersection can be accomplished.

Although each frustum consists of four corner rays, the frustum that results from their intersection can have between zero and eight corner rays. Similarly, each frustum has a rectangular base, but their intersection can have an arbitrary polygonal base. For example, Figure 3.37 shows the intersection of the bases of frusta **A** and **B**. In this case, the intersection, marked in red, forms another frustum with a rectangular base. In Figure 3.38, however, the intersection forms an irregular frustum with a polygonal base made up of eight corner rays.

These figures demonstrate the two ways in which intersection rays, the corner rays of the resulting frustum, are determined. Let **A** be the view frustum represented by

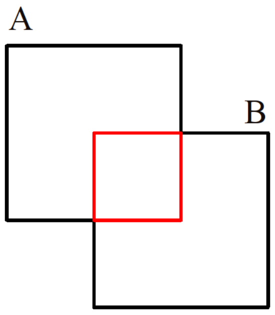


Figure 3.37: Intersection could result in a rectangular base.

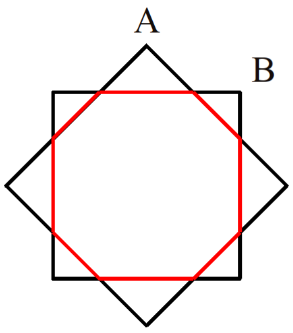


Figure 3.38: Intersection could result in a polygonal base.

the set of the view frustum's four corner rays, and let **B** be the frustum represented by the the set of the visible sphere frustum's four corner rays. First, intersection rays arise from the intersection of the planes of **A** with the planes of **B**. All of the intersection rays in Figure 3.38 are the result of plane-plane intersection. This is only possible if the two frusta share an apex, implying that all rays under consideration originate at this apex and that all frustum planes intersect at this apex.

An important property of this type of intersection ray is demonstrated in Figure 3.39. Given a particular intersection ray $\widehat{i}$ that results from the intersection of planes P_{A3} and P_{B4}, $\widehat{i}$ is a valid intersection ray if it is bounded by the opposite planes in both **A** and **B**. For **A** the bounding planes are P_{A2} and P_{A4}, and for **B** the bounding planes are P_{B1} and P_{B3}.

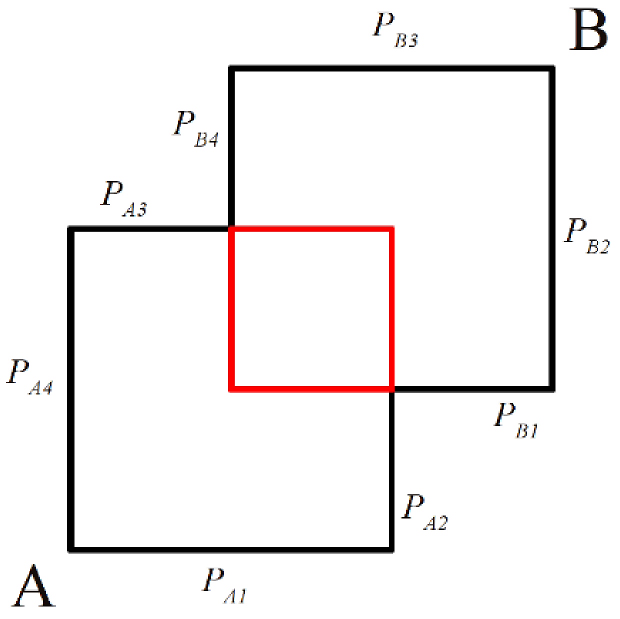


Figure 3.39: An example of bounding for intersection rays.

Figure 3.40 shows a case where the ray resulting from plane-plane intersection is bounded by the opposite planes of **A** but not the opposite planes of **B**. Therefore, this ray is not an intersection ray.

The second type of intersection ray is demonstrated in Figure 3.41. The intersection ray $\widehat{i_1}$ is a corner ray of **B** that exists within **A**, *i.e.* it is a ray that is bounded

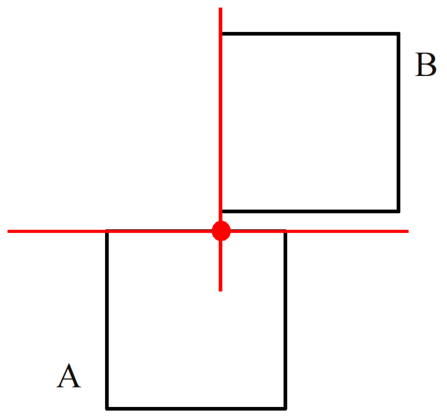


Figure 3.40: Plane intersection does not always yield an intersection ray.

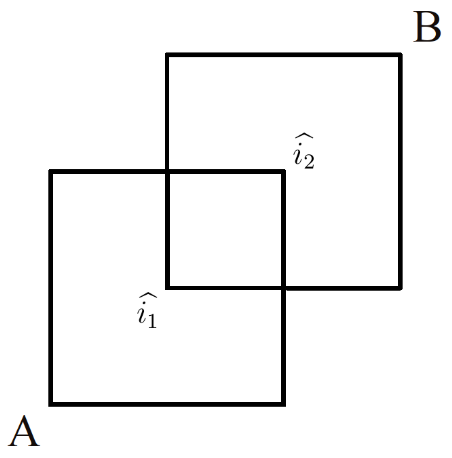


Figure 3.41: The corner ray of one frustum inside of the other.

by all four planes of **A**. Similarly, $\widehat{i_2}$ is a corner ray **A** that exists within **B**.

Before the algorithm can be described in detail, it is important to explain how a ray is determined to be bounded by a plane. A plane can be described by the normal to its surface. The angle between a ray and a plane is

$$\theta = \text{asin}(\widehat{r} \cdot \widehat{n}). \tag{3.7}$$

This is conceptually similar to Equation 3.9 in Section 3.6.3, which calculates the angle of an eye ray to the nadir. Instead of an up vector $\widehat{k}$, here we have the plane normal $\widehat{n}$ and the ray $\widehat{r}$. Note that the arcsine function outputs values in the range

$[-\frac{\pi}{2}, \frac{\pi}{2}]$. If the resulting angle θ is positive, then the ray is on the side of the plane that the normal points away from. This calculation can be simplified by noting that $\widehat{r} \cdot \widehat{n} > 0$ when the two vectors are acute, which signifies that the ray is on the normal's side of the plane.

Therefore, the algorithm that finds all of the intersection rays consists of four steps. All rays that are properly bounded are added to a list of intersection rays.

1. Calculate the normals of the planes in **A** and **B**.
2. Intersect the planes of **A** with the planes **B**.
3. Find the rays of **A** that exist within **B**.
4. Find the rays of **B** that exist within **A**.

The first step calculates the plane normals needed for determining whether a ray is bounded. For a particular frustum, we label the corner rays from $\widehat{c_1}$ to $\widehat{c_4}$ and the planes from P_1 to P_4, as shown in Figure 3.42. The plane normal $\widehat{n_1}$ corresponds to the plane P_1 and is found with the equation

$$\widehat{n_1} = \| \widehat{c_2} \times \widehat{c_1} \|.$$

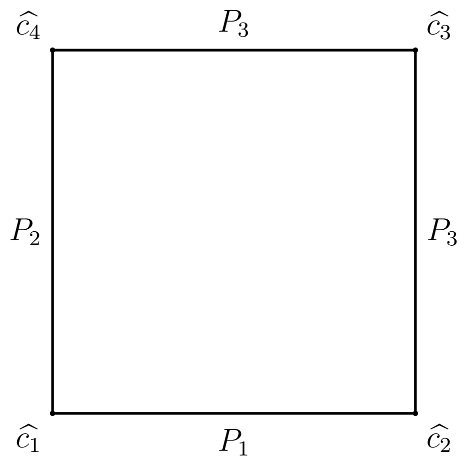


Figure 3.42: The ordering of planes and corner rays for an input frustum.

Similarly, $\widehat{n_2}$ is found using the cross product of $\widehat{c_3}$ and $\widehat{c_2}$, and so on. This pattern results in plane normals that point into the frustum, causing positive angles from Equation 3.7 for rays that exist inside the frustum.

Next, we intersect each plane of **A** with each plane of **B**. The intersection between two planes is the result of the cross product between the normals of the planes. A check is made to ensure that all rays are pointing in the negative z direction; if not, the ray is negated. This is necessary since the camera is directed along the negative z-axis, and all eye rays must point in this direction. Each of the resulting rays are then tested against the four opposite planes, two from **A** and two from **B**. If the ray is bounded by these planes, then it is added to the list of intersection rays.

The third and fourth steps are similar to each other. To find the corner rays of **A** that exist within **B**, each corner ray of **A** is tested against each plane of **B**. If the corner ray is bounded by all four planes, it is added to the list of intersection rays. A similar process is used to find the corner rays of **B** that exist within **A**. The list of intersection rays now represents the frustum resulting from the intersection of the two frusta **A** and **B**.

Finally, if there happens to be no intersection between the two frusta, as in the case that the viewer is looking directly away from the planet, we simply use the visible sphere frustum as the sampling camera. The disadvantage of this approach is that the terrain is oversampled.

3.5.4 Matrices

In the final step, we use the set of intersection rays to calculate the sampling camera's orientation matrix and projection matrix. The sampling matrix and its inverse are used to transform between world space and sampling camera space. The inverse of the projection matrix is used to generate eye rays from the grid points during PGM.

The sampling camera matrix **S** is formed from the concatenation of the view matrix **V**, the rotated eye matrix **R**, and the matrix **O** that transforms rotated eye space to sampling camera space. The equation

$$\mathbf{S} = \mathbf{O} \times \mathbf{R} \times \mathbf{V}$$

performs post multiplication in the correct order to produce S.

The matrix **O** is generated by building a coordinate system from the set of intersection rays. Specifically, the sampling camera's forward vector orients the near plane of the camera. The orientation of the near plane affects which eye rays are generated from the projective grid, which can negatively affect the distribution of eye rays. A near plane that leads to an even distribution of eye rays is given in Figure 3.43. The grid points are spread evenly across the frustum and result in a balanced set of eye rays. Alternatively, Figure 3.44 demonstrates a near plane that leads to an uneven distribution of eye rays. The grid points are distributed evenly across the near plane, but due to the near plane's tilt relative to the sampling camera, the set of eye rays is not balanced across all possible eye rays.

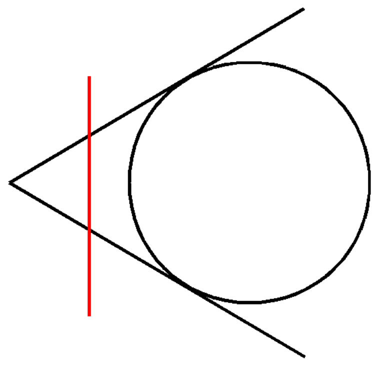


Figure 3.43: A good near plane leads to an even ray distribution.

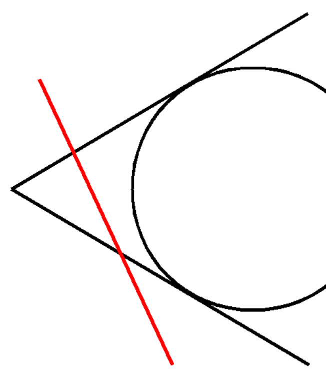


Figure 3.44: A bad near plane leads to an uneven ray distribution.

To obtain a near plane that well approximates the possible eye rays, we calculate the forward vector of the sampling camera as the average of the intersection rays. In addition, we must maintain the requirement that the sampling camera is oriented with the horizon, or more specifically, the yz-plane must include the center of the planet. Therefore, the y and z components of the forward vector are calculated as the average of the y and z components of the intersection rays, and the x component of the forward vector is set to zero.

With the new forward vector, we calculate the basis vectors that constitute the orientation of the sampling camera relative to rotated view space. As before, we use the nadir $\widehat{n}$.

$$\widehat{b} = -\widehat{f}$$

$$\widehat{r} = \|\widehat{b} \times \widehat{n}\|$$

$$\widehat{u} = \|\widehat{b} \times \widehat{r}\|$$

These vectors are used to construct the matrix **O** that transforms from rotated eye space to sampling camera space, allowing the sampling matrix **S** to be calculated.

Creation of the projection matrix requires the left, right, bottom, top, near, and far attributes for constructing a frustum. The near and far attributes are defined by the application programmer. The other attributes are found by intersecting the intersection rays with the sampling camera's near plane. Because the frustum attributes are defined relative to the sampling camera, we transform all of the intersection rays to sampling camera space before intersecting them with the near plane.

For each intersection ray $\widehat{c}$, the point of intersection on the near plane is determined by the intersection time t of the ray, which is calculated using the equation

$$t = -n/\widehat{c}_z.$$

This is a result of the law of similar triangles. The point of intersection P is then

$$P = t\,\widehat{c}.$$

The z component of each intersection point P is the near value, but the x and y components are coordinates on the near plane that are unique to each intersection ray. The minimum and maximum (x, y) components correspond to (left, bottom) and (right, top), which are the frustum attributes needed to build a projection matrix.

3.6 Projective Grid Mapping

The PGM algorithm operates mostly on the GPU. For a given grid point, an eye ray is generated. In order to work in the two-dimensional cross section, a coordinate system is required. This coordinate system consists of three basis vectors that describe points and vectors relative to the plane of the cross section.

Using this basis, we calculate the angle of the eye ray to the nadir. This ray angle is used to calculate the normal angle at the ray-circle intersection point. Once we have the normal angle, we must calculate the sphere normal in eye space. To accomplish this, we use a linear combination of the basis vectors calculated previously.

Finally, the normal angle is used to calculate the position of the ray-circle intersection. The position is generated in eye space to mitigate floating point precision issues that would result from measuring the position relative to world space, which has an origin at the center of the planet.

3.6.1 Grid Points to Eye Rays

The PGM algorithm first binds the two target buffers for rendering. These buffers hold sphere normals and positions in sampling camera space and are both 32-bit floating point RGB textures. The sampling camera matrix and the inverse of the sampling projection matrix are uploaded to the GPU. A fullscreen quad is rendered in order to generate a fragment for each pixel in the target buffers.

The rest of the algorithm takes place in the fragment shader. The first task is to find the eye ray associated with the current fragment. The current fragment coordinate, in space of pixels, ranges from $(0, 0)$ in the lower left corner of the screen to $(\text{width}, \text{height})$ in the upper right corner of the screen, where width and height are the dimensions of the target buffers. It is available in the fragment shader. The fragment coordinate is transformed into its clip space coordinate, which ranges from [-1, 1] on all axes and represents everything contained within the frustum. The equations for this are simple linear transformations.

The inverse of the projection matrix of the sampling camera is then used to transform the clip space coordinates into sampling camera coordinates. After the matrix multiplication, a perspective divide is performed, just as would be necessary if we were to use the projection matrix to transform sampling coordinates into clip coordinates. Since the origin of sampling coordinates is the eye, the position of the grid point in sampling coordinates is also a vector from the eye to the grid point. The position is seen as a vector and is normalized to get the eye ray for the grid point.

3.6.2 Cross Section Coordinate System

The next step is to calculate the local coordinate system that we will use to generate sphere normal vectors from the normal angle. This calculation begins with calculating the right vector from the cross product of the eye ray and the nadir, transformed into sampling camera space by the sampling camera view matrix. The cross product of the right vector with the nadir produces the up vector of the coordinate system. The nadir is taken to be the forward vector of the coordinate system.

We calculate the three basis vectors using the eye ray $\widehat{r}$ and the nadir $\widehat{n}$, both expressed in eye space. This coordinate system will be used to transform between coordinates of the cross section to coordinates of the viewing camera.

The three basis vectors of the coordinate system are $\widehat{\imath}$, $\widehat{\jmath}$, and $\widehat{k}$, corresponding to the x-, y-, and z-axes. The nadir $\widehat{n}$ is taken to be the first basis vector $\widehat{\imath}$. Next, the cross product of $\widehat{\imath}$ and $\widehat{r}$ produces the right vector $\widehat{\jmath}$. Finally, the cross product of $\widehat{\jmath}$ and $\widehat{\imath}$ produces the up vector $\widehat{k}$. The results of the cross products are normalized to obtain unit vectors, simplifying later calculations.

3.6.3 Ray Angle

Before calculating the normal angle, we must first calculate the ray angle ω. Working in the plane of the two-dimensional cross section, we use the dot product and a trigonometric identity.

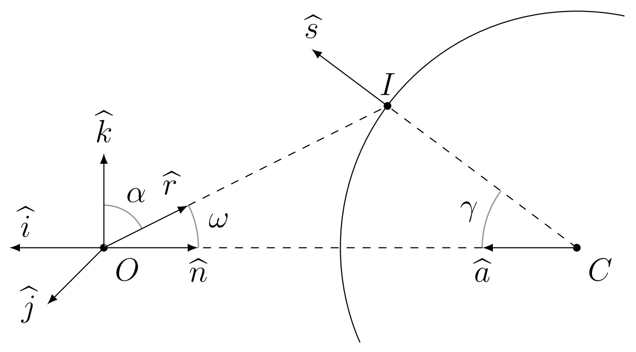


Figure 3.45: The calculation of the ray angle.

Figure 3.45 shows the angle ω between the ray $\widehat{r}$ and the nadir $\widehat{n}$, as well as the angle α between $\widehat{r}$ and the up vector $\widehat{k}$. Recall that $\widehat{k}$ and $\widehat{r}$ are unit vectors. Then the equation

$$\widehat{r} \cdot \widehat{k} = \cos\alpha \tag{3.8}$$

relates $\widehat{r}$ and $\widehat{k}$ to the angle α between them. Using the knowledge that $\widehat{k}$ is perpendicular to $\widehat{i}$, we note that

$$\omega = \frac{\pi}{2} - \alpha.$$

The trigonometric identity $\cos\alpha = \sin(\frac{\pi}{2} - \alpha)$ implies that

$$\cos\alpha = \sin(\frac{\pi}{2} - \alpha) = \sin\omega.$$

Substituting this result into Equation 3.8, we get

$$\widehat{r} \cdot \widehat{k} = \sin\omega,$$

allowing us to find

$$\omega = \text{asin}(\widehat{r} \cdot \widehat{k}). \tag{3.9}$$

Although the angle ω could be calculated using the dot product between $\widehat{r}$ and $\widehat{i}$, the arccosine function in most standard libraries outputs values in the range $[0, \pi]$. Our expression uses the arcsine function, which outputs values in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$. This signed range of values is more useful since it includes the angles of all possible rays that could intersect the circle. In addition, our use of arcsine results in an important simplification, described next.

3.6.4 Sphere Normal

Next we use Equation 3.3 and the ray angle to find the normal angle at the front facing intersection point for both reference spheres.

When Equation 3.3 is implemented on the GPU, the ratio $c = d/r$ can be precomputed. Additionally, let $u = \widehat{r} \cdot \widehat{k}$ where $\widehat{r}$ is the eye ray and $\widehat{k}$ is the up vector. Then substituting Equation 3.9 for ω in Equation 3.3 results in

$$\gamma_1 = \text{asin}(c\ \sin(\text{asin}(u))) - \text{asin}(u),$$

which reduces to

$$\gamma_1 = \text{asin}(c\,u) - \text{asin}(u). \tag{3.10}$$

Avoiding the sine function call improves execution time as well as precision since it results in less floating point operations.

Next, we calculate a blending factor to use to mix the two results of intersection with the two reference spheres. A user-defined threshold ζ is chosen to define the range of rays to use for blending. The threshold ranges from $[0, 1]$ and is a percentage of the total range of possible eye ray angles from the nadir.

The minimum value for a ray angle is 0, corresponding to the nadir, and the maximum value for a ray angle is equal to the tangent angle, which can be found using Equation 3.5. A user-defined constant ζ defines what percentage of the range of positive ray angles to use for interpolation. A value of $\zeta = 0.9$ corresponds to 90% of the range of positive ray angles. Let $b = \zeta\, \tau_0$ be the beginning angle of the range, where the result of interpolation is entirely from the primary reference sphere. The blending factor is calculated with the following equation:

$$f = (\omega - \tau_0)/(b - \tau_0) \tag{3.11}$$

where ω is the ray angle.

For example, if a given ray angle is 95% of the tangent angle with $\zeta = 0.9\%$, the resulting intersection point is the average of the ray-circle intersection between the primary and secondary spheres. If a given ray is equal to the tangent angle, the resulting intersection point will be only the intersection point found using the secondary reference sphere. A factor of zero corresponds to using only the primary reference sphere intersection, and a factor of one corresponds to using only the secondary reference sphere intersection.

With the blending factor, we calculate the ray used to intersect the secondary reference sphere. The factor is used to mix between the beginning ray angle and the angle of ray tangent to the secondary reference sphere. With the new ray angle, a ray vector is calculated. The dot product between the new ray and the up vector in the local coordinate system is taken. With this information, we can calculate the normal angle with the new ray and the secondary reference sphere.

To get the final normal angle, we use the blend factor again to mix between the normal angle from the primary reference sphere and the normal angle from the secondary reference sphere. The normal angle γ along with the up vector $\widehat{k}$ and the antinadir $\widehat{a}$ allow us to calculate the sphere normal $\widehat{s}$.

The expression $\cos\gamma$ represents the horizontal component of the sphere normal with respect to the antinadir, while $\sin\gamma$ represents the vertical component. Therefore, calculation of the sphere normal is accomplished using sine and cosine in a linear combination of the vectors:

$$\widehat{s} = \widehat{a}\,\cos\,\gamma + \widehat{k}\,\sin\,\gamma \tag{3.12}$$

If $\widehat{a}$ and $\widehat{k}$ are vectors in eye space, the resulting vector $\widehat{s}$ will also be a vector in eye space.

3.6.5 Position

Next, we generate the position of intersection from the normal. To maintain precision, the position is generated in sampling camera space as opposed to world space. World space positions suffer from floating point imprecision issues since the origin of the world frame is the center of the planet. Our method of generating positions in sampling camera space uses the law of sines. Consider Figure 3.46. The altitude a, ray $\widehat{r}$, ray angle ω, and normal angle γ are known. Although there are two reference spheres,

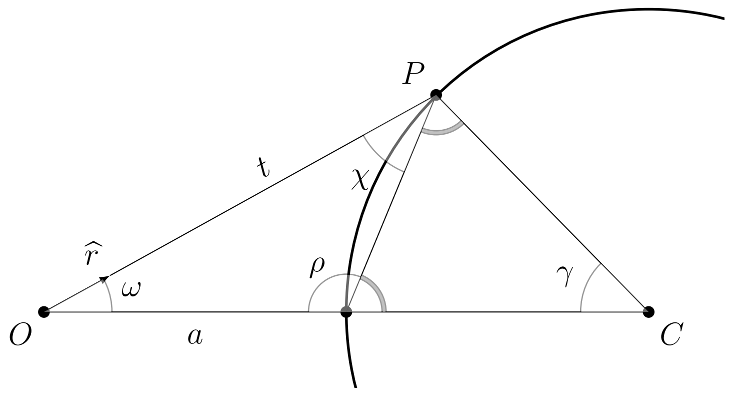


Figure 3.46: The intersection position in eye space.

the altitude a is the viewer's distance from the primary reference sphere. This does not make a difference for grid points that are a result of interpolating intersection results of the two reference spheres since the normal angle γ describes a position on the planet independent of either reference sphere. The angle χ is calculated from the angle γ using

$$\pi = 2\chi + \gamma$$
$$\chi = \frac{\pi}{2} - \frac{\gamma}{2}$$

when we observe that any triangle formed from an arc is an isosceles triangle. The angle ρ is then

$$\rho = \pi - \chi$$
$$\rho = \pi - (\frac{\pi}{2} - \frac{\gamma}{2})$$
$$\rho = \frac{\pi}{2} + \frac{\gamma}{2}$$

With the angle ρ, we can now calculate the distance t to the intersection point using the law of sines.

$$\frac{a}{\sin\chi} = \frac{t}{\sin\rho}$$
$$t = a\,\frac{\sin\rho}{\sin\chi}$$

The distance t and the eye ray vector $\widehat{r}$ are then used to calculate the position in sampling camera space.

$$P = t\,\widehat{r}$$

The sampling camera space vertex positions that result from this calculation are demonstrated in Figure 3.47. In the left image, the 16×16 projected mesh is rendered from the perspective of the viewer and without height displacement. The viewer is looking at the center of the planet. In the right image, the same mesh is shown in wireframe. A consequence of sampling camera space positions is that, when looking directly at the center of the planet, the grid is aligned with the screen.

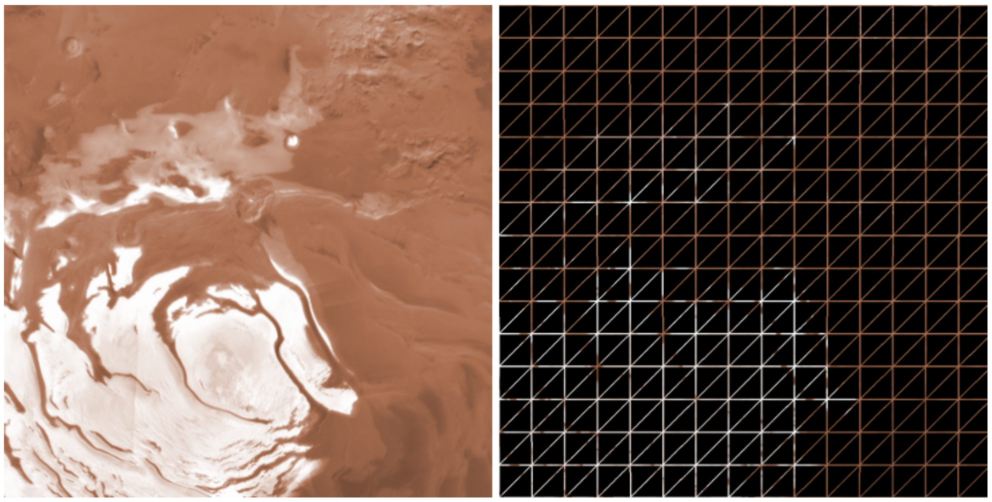


Figure 3.47: The vertex positions in sampling camera space.

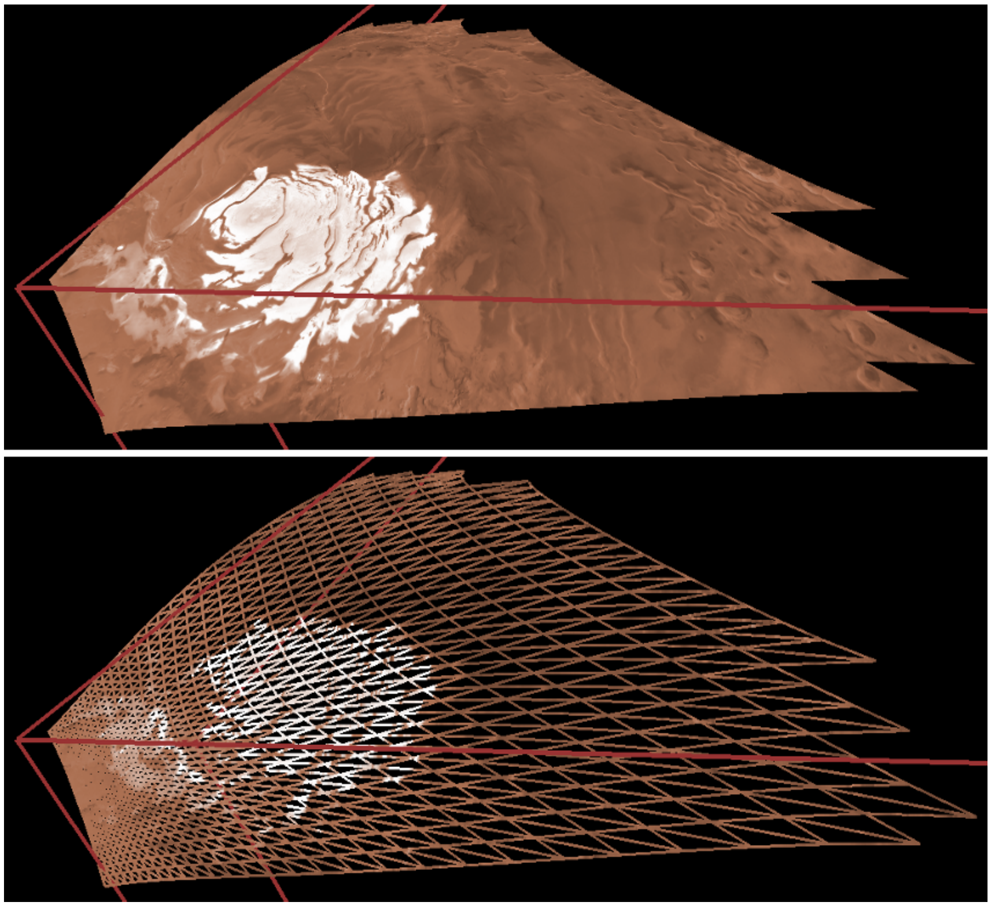


Figure 3.48: A 16 × 16 projected grid.

Figure 3.48 shows the result of the PGM step within the visualization. The projected mesh is rendered along with the sampling camera, marked in red. The top image shows the mesh with fully textured polygons, while the bottom image shows the mesh in wireframe. Note that the size of the polygons gradually increases as the distance from the viewpoint increases, which corresponds to a gradual LOD transition. Also, the projected mesh is curved on its sides due to the interpolation of intersection results from the primary and secondary reference spheres.

3.7 Deferred Texturing

Deferred texturing comprises the last four steps of the algorithm, and its purpose is to composite data over the geographic area of the projected grid. We implement the deferred texturing algorithm given by Kooima [33] with a few differences, described in this section. A figure accompanies each step, demonstrating the step's visual results. Figure 3.49 illustrates the end result. The left image shows the terrain with atmosphere and lighting. The camera is pointed up and right from the center, as demonstrated in the right image, which shows the sampling camera rotated slightly clockwise to match the viewing camera's direction.

The output of PGM consists of two buffers: the sphere normals and the positions, as shown in Figure 3.50. The sphere normals in the left image and the positions in the right image are scaled to the range $[0, 1]$ in order to be displayed as colors. The sphere normals buffer contains the sphere normal at each grid point, and the positions buffer contains the position of each grid point. In these buffers, the sphere normals and positions are represented with respect to sampling camera space.

The sphere normals are used to sample height and color maps because each sphere normal corresponds to a position in geographic coordinates. The positions are used for rasterizing the projective grid as a mesh.

These buffers have the same dimensions as the projective grid since there is a one-to-one relationship between grid points, sphere normals, and positions. Also, note that the dimensions of the projective grid are not necessarily the same as the

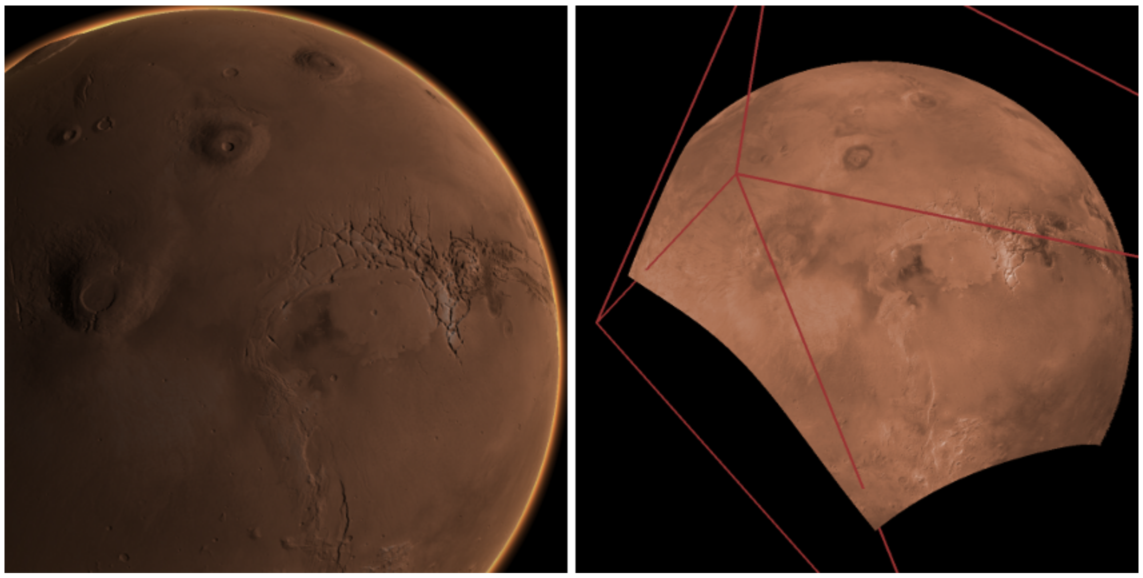


Figure 3.49: The goal of deferred texturing.

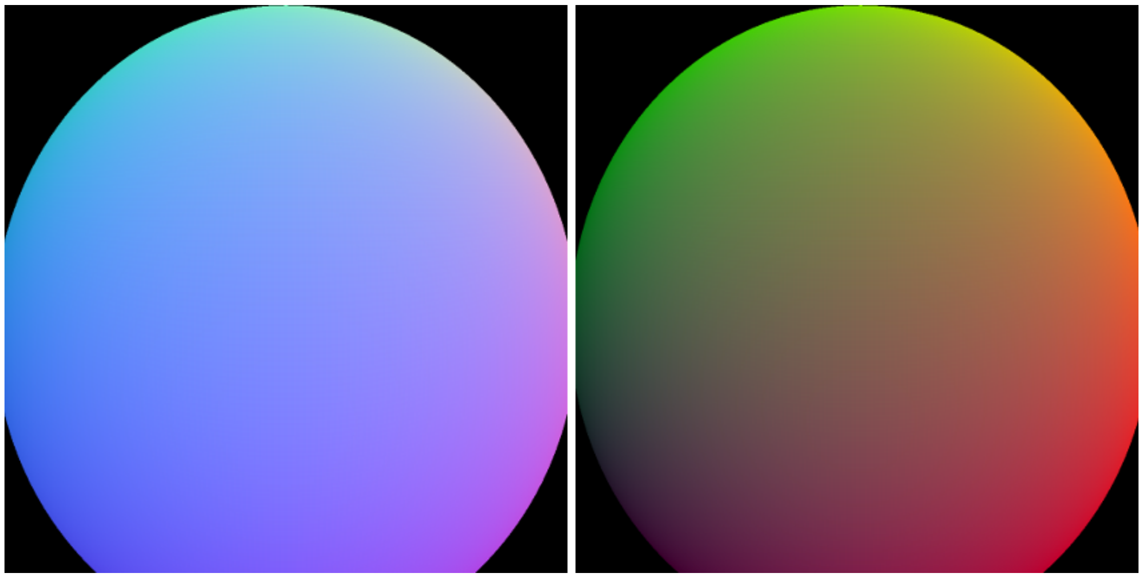


Figure 3.50: The sphere normal and position buffers from the PGM step.

dimensions of the screen, an observation necessary for a later optimization. Finally, observe that the shape of the planet in Figure 3.49 does not match the shape of the planet in Figure 3.50. This is because the buffers from PGM match the sampling camera, not the viewing camera. In addition, the planet appears elongated in the

PGM buffers since the sampling frustum has a rectangular base, as opposed to the viewing frustum which has a square base.

3.7.1 Height Composition

The goal of height composition is to iterate over all the height maps and combine the height values for each grid point. The accumulated height values are stored in a buffer of 32-bit floating points that has the same dimensions as the projective grid. On the CPU, one composition pass occurs for each height map. During a pass, the current height map is bound for reading on the GPU. Next, the constants used for the projection equations are uploaded to the GPU. The projection equations and the calculation of these constants are described in Section 4.1.

Every composition pass needs to read from and write to the accumulation buffer; on the GPU, the current accumulation value is mixed with the value from the current height map and written back to the accumulation buffer. Because a texture cannot be read from and written to at the same time, a technique called ping ponging must be used. In this approach, two buffers are used instead of one. The buffers alternate between being the input buffer and the output buffer. Before each pass, the buffers swap roles, and the output buffer from the last step becomes the input buffer for the next step, while the input buffer from the last step becomes the output buffer for the next step.

After the accumulation buffers switch roles, a full screen quad is rendered in order to trigger a fragment for each grid point. The rest of the algorithm takes place on the GPU.

Each fragment corresponds to a grid point. The first step for the fragment shader is to convert the grid point's sphere normal to geographic coordinates so that the currently bound height map can be sampled. Given a sphere normal $\widehat{n}$, the equations for geographic coordinates are

$$\lambda = \operatorname{atan2}(\widehat{n}_x, \widehat{n}_z),$$
$$\phi = \operatorname{asin} \widehat{n}_y.$$

The standard implementation of atan2 results in a longitude λ in the range $[-\pi, \pi]$. Because the projection equations require $[0, 2\pi]$ for longitude, we add π to shift the range of λ. The latitude ϕ is in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$ as a result of the equation, and this is the range required for the projection equations. A sphere normal equal to the zero vector represents an eye ray that missed the planet. In this case, the value from the input buffer is copied to the output buffer, and the fragment shader ends execution. Note that PGM outputs the sphere normal in sampling camera space; before being converted to geographic coordinates, the sphere normal must be transformed into world space.

Next, the geographic coordinates are converted to texture coordinates using the projection equations appropriate for the current map. If the position described by the geographic coordinates lies within the geographic area of the current map, the resulting texture coordinates are in the range $[0, 1]$. Alternatively, if the position does not lie within the geographic area of the height map, then it should not be sampled. In this case, the shader ends execution similarly to when the sphere normal is equal to the zero vector.

If the position lies within the geographic area of the height map, the height map is sampled. A minimum and maximum height value are defined for each height map; if the sampled value is outside the range of values for the current height map, the shader ends execution. Height data comes in a variety of data types and units, so we choose to convert ahead of time all height maps to 32-bit floating point values. For the purpose of our visualization of Mars, the units are in meters relative to Mars's average equatorial radius of $3,396,000$ m, since this is the convention that the MOLA dataset uses [43].

Next, the sampled height value is combined with the input height value depending on the composition operation specified for the current height map. We define three operations: average, replace, and add. The average operation is used for most data, especially in the case of overlapping height and color maps. The replace operation is used for the first composition pass in order to initialize the output accumulation buffer, which will be used as input in the next pass. Finally, the add operation is used for the areoid (or geoid in the case of Earth). The global topographic heights provided by the MOLA dataset are measured as offsets from the areoid height at their geographic locations. When the areoid height map, also provided by MOLA, is added to the topographic heights, the correct planetary shape is obtained. In contrast to our simple composition operations, Kooima [33] uses a sophisticated weighting function based on the map projection.

The output buffer for the last pass is used as the accumulated heights buffer. This buffer is shown in Figure 3.51, with the height values scaled to the range $[0, 1]$ for display as colors. Note that features such as mountains and canyons are rotated counterclockwise in relation to their appearance in the final image, shown in Figure 3.49. This is again a result of the sampling camera being rotated clockwise to fit the horizon in the direction of the viewing camera.

3.7.2 Rasterization

The sphere normals buffer, positions buffers, and accumulated heights buffer describe the final terrain mesh. At this point, we could proceed to composite colors and normals into buffers with the same dimensions as the previous three buffers; however, in order to decouple the performance of PGM from the performance of deferred texturing, we rasterize the grid-sized buffers into screen-sized buffers, an optimization described by Kooima [33]. This optimization accomplishes performance throttling by allowing grid size to differ from screen size. In addition, if parts of the projective grid lie outside of the screen or the projective grid is larger than the screen, this decoupling leads to a definite speed improvement since there are ultimately fewer fragments to process working with screen-sized buffers than with grid-sized buffers.

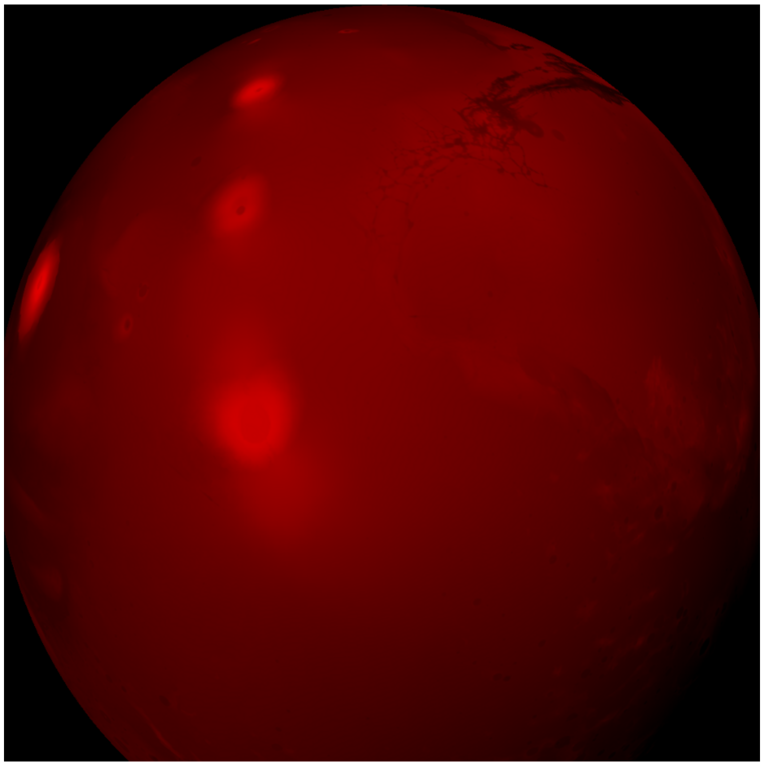


Figure 3.51: The accumulated heights buffer displayed as a red color channel.

In order to use the three buffers as geometry, we render a triangle strip where each vertex corresponds to a grid point in the projective grid, or alternatively, a texel in one of the three buffers. Each vertex in the triangle strip has an (x, y) position equal to the corresponding grid point's texel coordinates within the three buffers.

The vertex shader reads the position of the incoming vertex and determines which texel to read from each of the three buffers. The resulting sphere normal, position, and height value are used to determine the vertex's final position. The position is displaced along the sphere normal scaled by the height value. Before the height value can be used, it must be converted to units of the visualization. As noted in the previous section, this height value is in units of meters relative to the average equatorial radius $3,396,000$ m. We convert this value to kilometers and apply an optional scaling value. In addition, another offset must be accounted for. The position of the grid point is calculated as the intersection of an eye ray and the primary reference sphere (Section 3.6.5), and therefore the position's distance from the center of the planet is the primary reference sphere's radius. However, the accumulated height value (in kilometers) is relative to the average equatorial radius $3,396$ km, not the primary reference sphere's radius. Unless the primary reference sphere's radius is equal to $3,396$ km, we must account for the difference between these radii. Given the input height value h, the scaling value s, and primary reference sphere radius r, the offset to use for calculating the final position is

$$o = h \times (\frac{1 \text{ km}}{1000 \text{ m}}) \times s + (3396 \text{ km} - r).$$

Using the sphere normal $\widehat{n}$, position P, and offset o, the equation for the final position of the vertex is then

$$V = P + o \times \widehat{n}.$$

Because both the sphere normal and the position are in sampling camera space, the final position is a point in sampling camera space.

Next, the vertex shader outputs the sphere normal, final position, and height value. These three values are interpolated and sent by the fragment shader to the appropriate output buffer for rasterization. The sphere normal is transformed to world space since it is no longer needed in sampling camera space. The final position is transformed to eye space, since this is required later for the lighting equations. The height value is rasterized in order to perform minimum mipmapping (Section 3.4.1) for the purpose of finding the primary reference sphere radius to use in the next frame.

The interpolation that occurs in the rasterization step is the motivation for using the sphere normal to sample the datasets in the height and color composition steps, as opposed to having the PGM step output a geographic coordinates buffer. Although using a geographic coordinates buffer would save having to convert the sphere normal multiple times, the interpolation of geographic coordinates leads to errors across the prime meridian where longitude goes from 359° to 0°.

This situation is demonstrated in Figure 3.52. In the left image, longitude is rendered for each terrain point, where green corresponds to 0° longitude and yellow corresponds to 360° longitude. In the right image, the terrain is textured according to these longitude values. The correct result would be that interpolated longitude values instantaneously change to 0° as soon as 360° is exceeded. However, because triangles lie across the prime meridian, one vertex is close to 360° and another vertex is close to 0°, and interpolating between these two values leads to a longitude transition from 360° through 180° and downward to 0°.

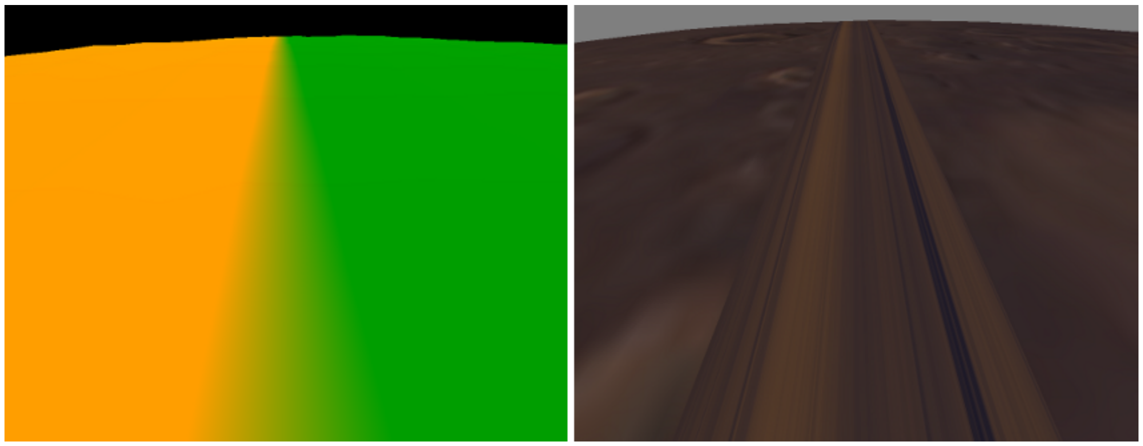


Figure 3.52: Incorrect interpolation of longitude across the prime meridian.

This problem is solved by interpolating sphere normals across the prime meridian. Because sphere normals are three dimensional, there is no sudden transition as in longitude. The next step, color composition, can then convert the sphere normals to geographic coordinates and correctly sample the datasets. Note that it is possible to use a grid-sized geographic coordinates buffer generated during PGM and a screen-sized geographic coordinates buffer generated in the fragment shader during rasterization. This would save having to recompute geographic coordinates from the sphere normal. However, this incurs an additional memory cost on the GPU.

The output sphere normal and position buffers are shown in Figure 3.53. Due to rasterization, the shape of the planet is apparent in these buffers; note the mountain on the horizon in the upper left corner and the slight depression of the canyon on the right side. The silhouette of the planet now matches that of the final image in Figure 3.49 since these buffers are in screen space and show the planet relative to the viewing camera.

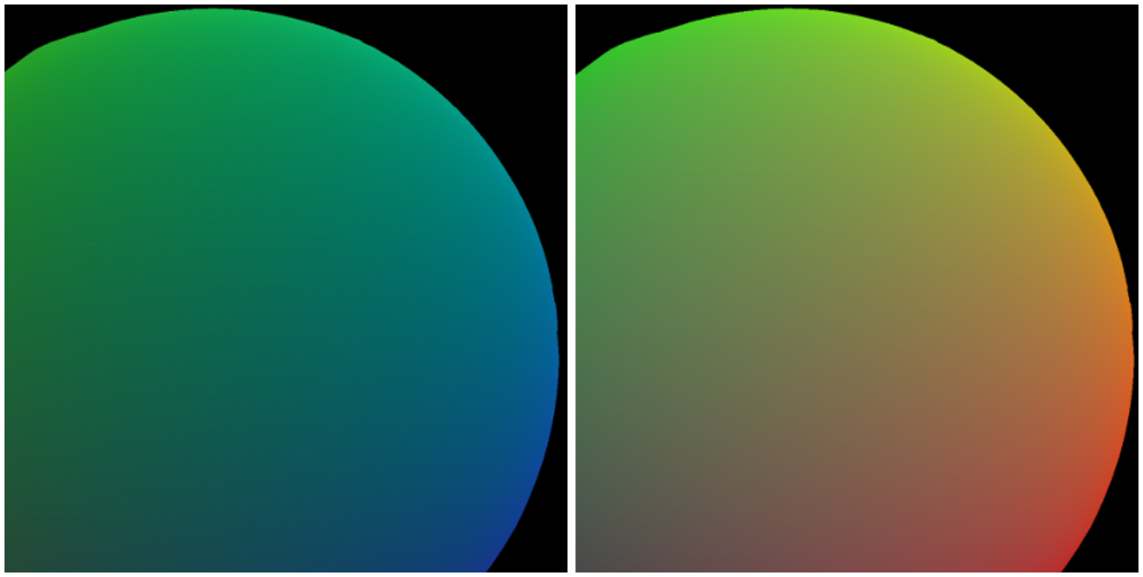


Figure 3.53: The screen-sized sphere normal and position buffers.

3.7.3 Color Composition

Color composition is similar to height composition. Because colors and surface normals use three components, the same CPU and GPU code can be used for each of these types of data, and we perform the composition for colors and surface normals together. The normal maps composited in this step are generated from the height maps beforehand (Section 4.2). The accumulated color and surface normal buffers are shown in Figure 3.54.

3.7.4 Atmosphere and Lighting

The final step is to render the outer atmosphere and combine the accumulated color and surface normal buffers using standard Phong lighting equations with atmospheric effects. The shaders for rendering the outer atmosphere and adding atmospheric effects to the terrain are given by Sean O' Neil [51]. Figure 3.55 shows the result of this step.

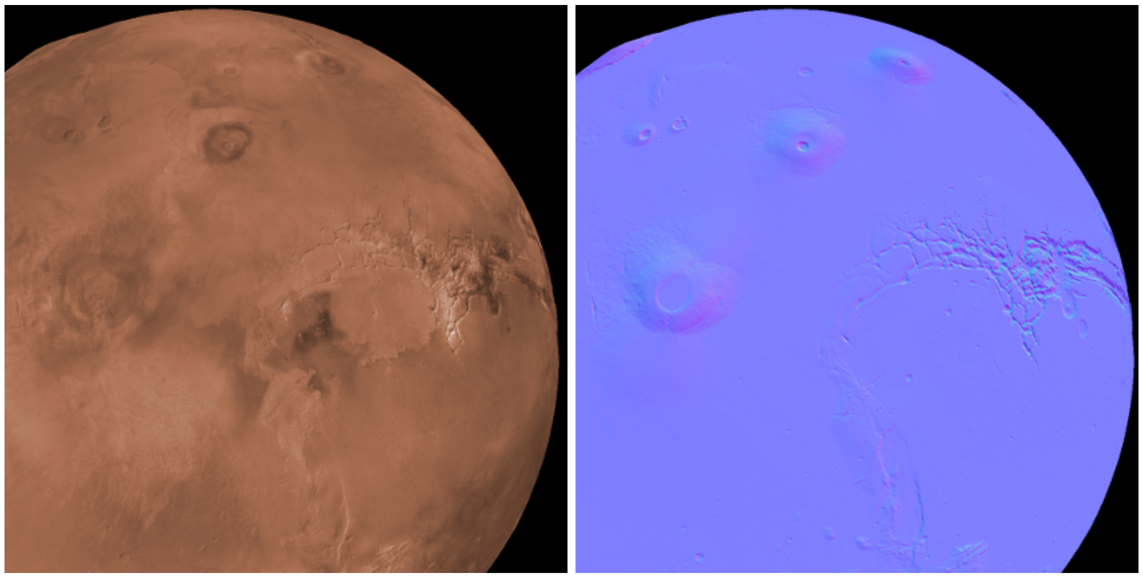


Figure 3.54: The accumulated color and surface normal buffers.

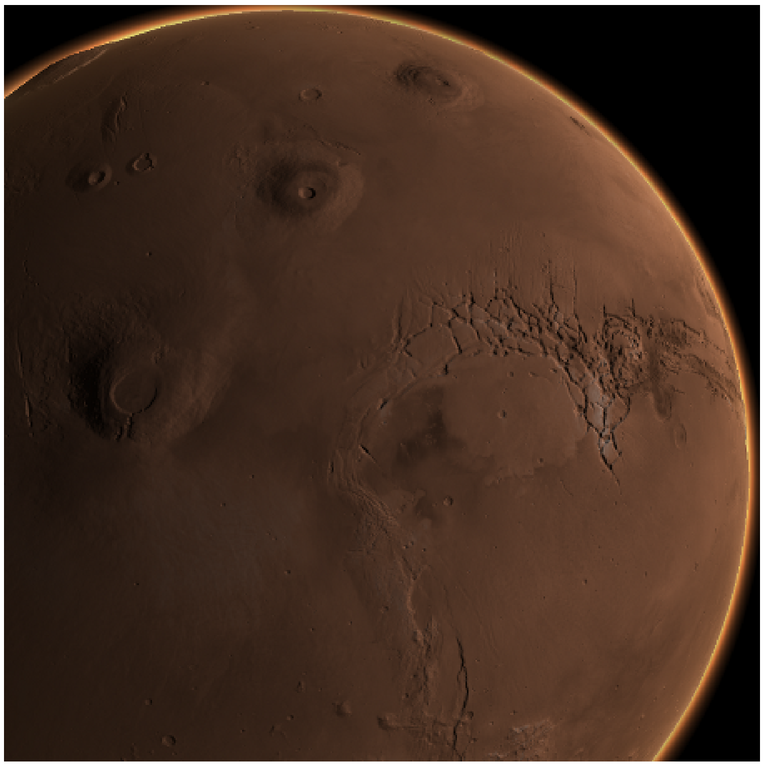


Figure 3.55: The terrain with atmosphere and lighting applied.

Chapter 4

Data Processing

In order for deferred texturing to work, a number of operations on the data must occur before and during the simulation. In this chapter, we cover three such operations.

First, during the height and color composition steps, geographic coordinates must be converted to texture coordinates using projection equations on the GPU. Second, the atmosphere and lighting step requires surface normals generated from the original height maps. Finally, after the terrain has been rendered, other objects may need to be rendered along with the terrain at near and far distances. The scale of the planet prevents the use of a single depth buffer; therefore, a method must be devised to correctly occlude objects with the terrain.

4.1 GPU Map Projections

The map projection equations are required for the height composition step and the color composition step, both of which occur on the GPU using the fragment shader. In addition, the map projection equations given by Eliason [18] involve multiplication by either the equatorial radius or the polar radius. This is unsuitable for the GPU since planetary radii are large numbers and floating point precision is limited. Moreover, no equations are given by Eliason to obtain texture coordinates since this is a specific need of graphics applications. Therefore, we reformulate the map projection equations given for implementation on the GPU. First, we describe the various coordinate systems and transformations that are required to go from geographic coordinates to texture coordinates. Next, we describe how these transformations are calculated from the Planetary Data System (PDS) label information (described in Section 2.4.3) for each dataset and how the transformations are implemented on the GPU. Methods for both the equirectangular projection and the polar stereographic projection are given.

4.1.1 Coordinate Systems

In either of the composition steps, the input texture is queried for each projected grid point's geographic location. This location is encoded as a pair of geographic coordinates (λ, ϕ) corresponding to longitude and latitude. In order to read the texture, these coordinates must be converted to texture coordinates. There are four coordinate systems involved in this transformation: geographic coordinates, projection coordinates, map coordinates, and texture coordinates.

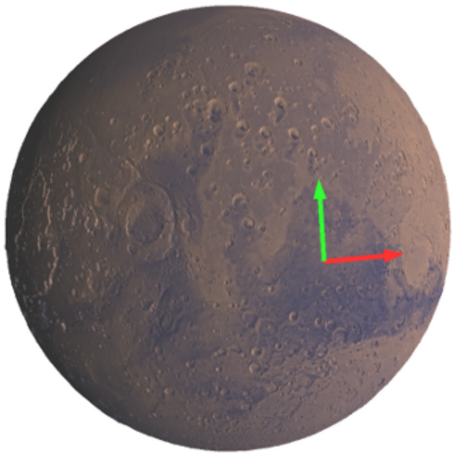


Figure 4.1: Geographic coordinates encode positions on the sphere.

The origin of geographic coordinates exists at the intersection of the equator and the prime meridian, as shown in Figure 4.1. The red axis represents longitude and the green axis represents latitude. Longitude values are in the range $[0°, 360°]$ and latitude values are in the range $[-90°, 90°]$. The output of PGM is a sphere normal for each grid point; during either composition step, the sphere normal is converted to geographic coordinates, and the longitude and latitude values are ensured to be within these ranges.

The geographic coordinates for a grid point must be transformed into the projection of the texture to be queried. This new coordinate system is called projection coordinates. The transformation from geographic coordinates (λ, ϕ) to projection coordinates (p, q) depends on the type of map projection used to create the texture.

The origin of projection coordinates, called the center of projection, is independent of the geographic extent of the texture. Instead, the origin depends on the map projection used and is usually chosen to be at the equator, the north pole, or the south pole depending on the map projection. Figure 4.2 shows a possible center of projection for the equirectangular projection. The red axis represents the x-coordinate and the green axis represents the y-coordinate.

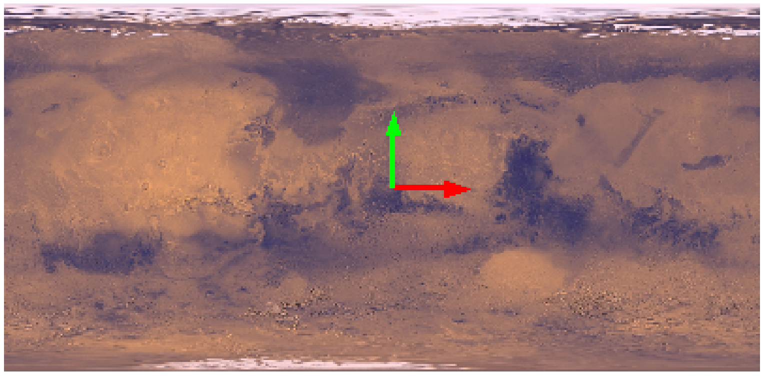


Figure 4.2: The center of projection for an equirectangular projection.

Projection coordinates are converted to map coordinates (x, y) based on the position of the center of projection relative to the origin of map coordinates. Map coordinates originate in the upper left corner of the image, as shown in Figure 4.3. The scale of map coordinates is the same as that of projection coordinates; only the origin differs. The concept of map coordinates is useful for obtaining texture coordinates.

The range of texture coordinates is $[0, 1]$ in both dimensions. Let the point D be the lower right corner of the image in map coordinates, which represents the width

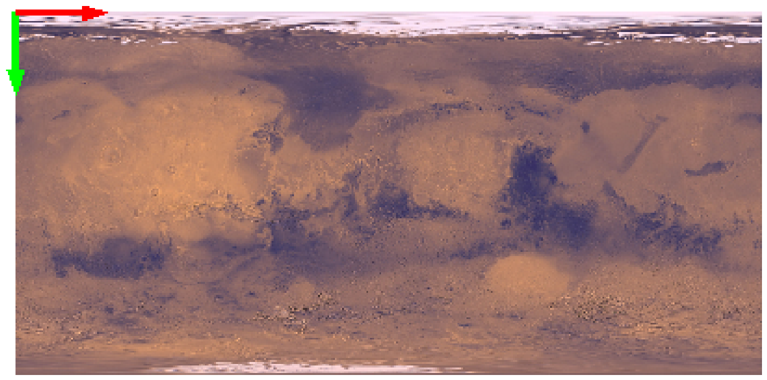


Figure 4.3: Map coordinates originate in the image's upper left corner.

and height of the image since the origin is the upper left corner. Any point in map coordinates can be converted to texture coordinates by dividing by the components of D.

The origin of texture coordinates is commonly chosen to be the lower left corner of the texture. In this case, the horizontal component of texture coordinates remains the same as that of map coordinates, but the vertical component is reversed, as shown in Figure 4.4. To remedy this, the vertical component needs to be subtracted from one before being used as a texture coordinate. However, if the origin of texture coordinates is chosen to be the upper left corner of the texture, the texture coordinates can be used directly after conversion from map coordinates.

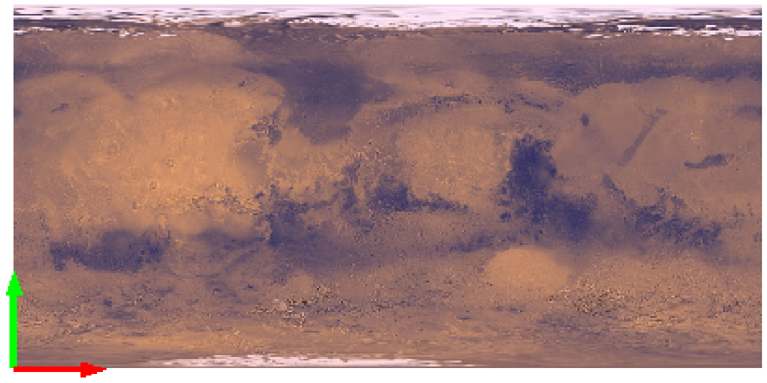


Figure 4.4: Texture coordinates typically originate in the lower left corner.

We will now describe the process of converting from geographic coordinates to texture coordinates for both the equirectangular projection and the polar stereographic projection, including the equations implemented on the GPU and the constants in these equations that need to be uploaded to the GPU.

4.1.2 Equirectangular Projection

The transformation from geographic coordinates to projection coordinates is accomplished using the forward mapping equations. The forward mapping equations for the equirectangular projection, given by Eliason [18], are

$$p = R\,(\lambda - \lambda_0)\,\cos\phi_0,$$

$$q = R\,\phi,$$

where λ is longitude, ϕ is latitude, λ_0 is the longitude of the center of projection, ϕ_0 is the latitude of the center of projection, and R is the radius at the center of projection. We simplify these equations by removing the multiplication by R. Although this changes the scale of projection coordinates, it does not affect the geographic area that can be represented. This simplification removes the need to multiply by a large number on the GPU, improving precision.

The center of projection (λ_0, ϕ_0) for an image with a PDS label can be obtained from the center latitude and center longitude fields for λ_0 and ϕ_0, respectively. We convert these values from degrees to radians for use on the GPU, where the trigonometric functions operate with angles in radians.

The transformation from projection coordinates to map coordinates is accomplished using the equations

$$x = p + C_x,$$

$$y = -q + C_y,$$

where (C_x, C_y) is the center of projection in map coordinates. The vertical component is negated since the vertical axis of map coordinates points in the opposite direction of the vertical axis of projection coordinates.

The point (C_x, C_y) is obtained from the sample projection offset, line projection offset, and map resolution fields of the PDS label. The sample projection offset S and the line projection offset L represent the center of projection in pixel coordinates (i, j) of the image. The PDS convention is that $(0.5, 0.5)$ represents the upper left corner of the upper left pixel, however in computer graphics $(0, 0)$ is commonly used. Although it is given in pixel coordinates, the center of projection is not necessarily a position within the image. The map resolution is the scale of the map at the center of projection and is given in units of pixels/degree; however, we convert to units of pixels/radian. The scaling value M is the inverse of the map resolution, which is in units of radians/pixel. This allows the conversion of pixel coordinates to map coordinates by multiplying by M. The equations for C_x and C_y are then

$$C_x = (S - 0.5) * M,$$
$$C_y = (L - 0.5) * M.$$

The sample and line projection offsets are subtracted by 0.5 so that the position of the upper left corner of the upper left pixel is $(0, 0)$.

The transformation from map coordinates to texture coordinates requires the location of the lower right corner in map coordinates (M_x, M_y), which will be used to scale the input map coordinates to the range $[0, 1]$ for texture coordinates. This point is obtained using the width W and height H of the texture in pixels, which is available in the PDS label. We multiply these dimensions by the scaling value M, which is similar to how the center of projection was converted to map coordinates. This is demonstrated in the equations

$$M_x = W * M,$$
$$M_y = H * M.$$

On the GPU, we convert map coordinates (x, y) to texture coordinates (s, t) using the equations

$$s = x/M_x,$$
$$t = y/M_y.$$

For efficiency, the inverses of M_x and M_y are calculated on the CPU and uploaded to the GPU.

4.1.3 Polar Stereographic Projection

We modified the forward mapping equations for the polar stereographic projection from Eliason [18] to allow the center latitude (either 90° for the north pole or −90° for the south pole) to affect whether the equations produce projection coordinates for a south or north polar stereographic projection. The forward mapping equations are

$$p = c\,\sin(\lambda - \lambda_0),$$
$$q = -c\,s\,\cos(\lambda - \lambda_0),$$

where $s = \sin(\phi_0)$ is the scaling value equal to -1 for the south pole or 1 for the north pole and c is a factor similar to both equations and calculated using

$$c = 2\tan(\frac{\pi}{4} - \frac{s\phi}{2}).$$

Note that the scaling value s is used in the calculation of c. As in the equirectangular projection, we omit the multiplication by the polar radius, which is usually present in polar stereographic projection equations. Without this multiplication, the range of projection coordinates is essentially the unit circle, where c determines the distance from the center of projection and the sine and cosine components represent the rotation from the line longitude 0°, the prime meridian.

The coordinate system of projection coordinates relative to the image is shown in Figure 4.5. In this example, topographical data from MOLA [43] for Mars' north pole is used. The center of projection in this case is the center of the image. Longitude 0° extends downward from the center, and longitude 90° extends to the right. For

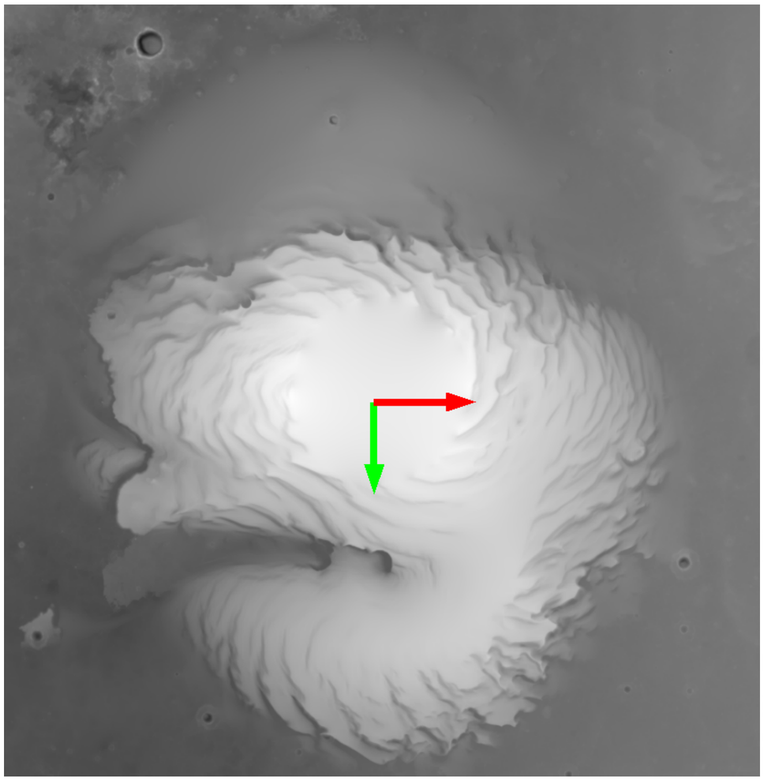


Figure 4.5: The coordinate system of a north polar stereographic projection.

a south polar stereographic projection, longitude 0° extends from the center toward the top of the image and longitude 90° extends to the right.

The transformation to map coordinates is similar to the equirectangular projection, except a different scale value M is used to calculate the center of projection (C_x, C_y) in map coordinates. The map scale field of the PDS label is used instead of the map resolution field. The map scale field is similar to the map resolution field except that it encodes the scale of the map in units of kilometers per pixel. The scale value M is calculated as C/R, where C is the value of the map scale field and R is the polar radius. The center of projection is then calculated as before, and projection coordinates are transformed into map coordinates using the same equations as before.

The transformation to texture coordinates is similar to the equirectangular projection except that the calculation of the lower right corner (M_x, M_y) in map coordinates uses the special scale factor described previously.

4.2 Normal Map Generation

Accurate lighting requires surface normals. Although surface normals could be calculated at runtime from the projected and displaced mesh, the PGM-generated mesh tends to vary slightly as the camera moves. Instead, we precompute surface normals from each of the input height maps in a separate preprocessing step. This program uses the fragment shader to quickly build the normal map and then writes the normal map to an image file. The normal map has the same dimensions as the height map used to create it. In this section, we will describe the process used for calculating normals for both projections.

4.2.1 Normals from Equirectangular Data

To encode normals in the normal map, we use the standard coordinate system for tangent space where is x is longitude, y is latitude, and z is up. This coordinate system is the same used for the equirectangular projection, shown previously in Figure 4.2.

To calculate the surface normal for a pixel (i, j), we use a method similar to Angel [1, 492-495]. The normal is based on the surface defined by the neighboring four height values of (i, j). Specifically, the normal is calculated from the cross product of two vectors. The first vector is the tangent and goes from the left texel $(i-1, j)$ to the right texel $(i+1, j)$. The second vector is the bitangent and it goes from the bottom texel $(i, j-1)$ to the top texel $(i, j+1)$. We take the cross product of the tangent and the bitangent and normalize the result to obtain the normal for pixel (i, j).

4.2.2 Normals from Polar Stereographic Data

The tangent coordinate system matches the equirectangular projection in that the tangent basis vector points in the direction of positive longitude and the bitangent basis vector points in the direction of positive latitude, that is, toward the north pole. However, in the polar stereographic projection, the top of the image no longer corresponds to the north pole. Instead, the center of projection may be located anywhere, although commonly it is in the center of the image as in Figure 4.5.

In the case of the polar stereographic projection, all calculated normals need to be rotated into the same tangent coordinate system as that used for the equirectangular projection. Although this could be performed on the GPU at runtime, it is more efficient to do this rotation directly after the normal is computed and store the rotated normal inside of the normal map.

The normal is calculated using the same method used for the equirectangular projection. Next, a set of three basis vectors representing the tangent coordinate system at the pixel (i, j) is constructed. First, the bitangent, the basis vector corresponding to the y direction in an equirectangular projected image, is calculated as the vector pointing from the pixel (i, j) towards the center of projection. The center of projection is obtained from the PDS label as before. The tangent basis vector, corresponding to the x direction, is calculated as the cross product of the bitangent and the up vector $(0, 0, 1)$, which corresponds to the sphere normal at pixel (i, j). The tangent, bitangent, and up vector are placed into a rotation matrix. The computed normal is post multiplied with the rotation matrix to rotate it into the tangent coordinate system used by the equirectangular projected normal maps.

Figure 4.6 demonstrates the tangent coordinate system for four points in a polar stereographic projection. The up basis vector is not shown since it points directly out of the figure. The center of projection is in the center of the image, and it corresponds to the north pole. Notice that for each point, the bitangent, marked in green, points towards the north pole as it would in the equirectangular projection, demonstrated previously in Figure 4.5.

4.3 Scene Composition

A typical obstacle in rendering a planet to scale is the limited precision of the depth buffer, which is used for occlusion testing. When terrain is rendered at both a close distance (*e.g.* 1 meter) and a far distance (*e.g.* $10,000$ kilometers), the 24- or 32-bit

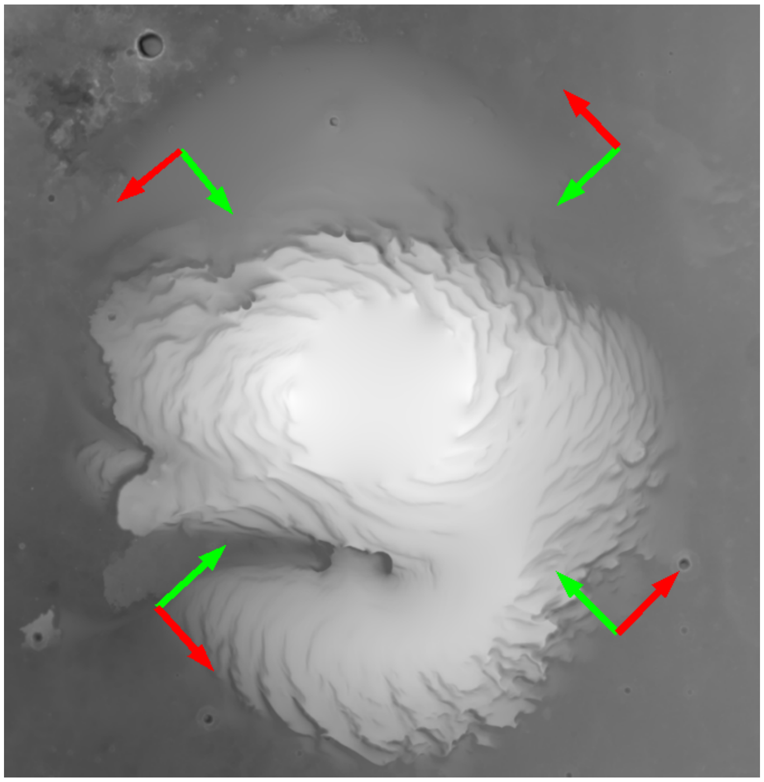


Figure 4.6: The tangent coordinate system in a polar stereographic projection.

floating point values used to store the depth of each fragment are insufficient for accurate occlusion testing, leading to a flickering artifact known as z-fighting.

To solve this problem, we use a multipass approach similar to Brandstetter [3]. The terrain is rendered twice with two different frusta: the near frustum and the far frustum. First, the close regions of the terrain are rendered by setting the current frustum to the near frustum, which has a near plane distance on the order of 1 centimeter and a far plane distance on the order of 1 kilometer. Second, the distant regions of the terrain are rendered by setting the current frustum to the far frustum, which has a near plane distance on the order of 1 kilometer and a far plane distance on the order of $10,000$ kilometers. Depth values are written to a unique depth buffer for each pass.

Although this method solves the problem, it presents a challenge with respect to rendering other objects along with the terrain. An object falls within either the near frustum or the far frustum, and it should be occluded correctly with the terrain in either region.

We present a method of rendering other objects after the terrain that uses both depth and stencil buffers to correctly determine which fragments of the other objects are occluded by the terrain at both near and far distances. During the rasterization step, information for the near terrain and far terrain is written into depth and stencil buffers. After the atmosphere and lighting step, the framebuffer contains the final image of the terrain, and other objects are rendered over the terrain with proper occlusion testing by using the depth and stencil buffers.

4.3.1 Rasterization

We now describe how the depth and stencil buffers are written for each pass of the rasterization step. The terrain is first rendered with the near frustum, near depth buffer, and near stencil buffer enabled. Whenever a fragment is written, a value of 1 is written to the stencil buffer in order to mark the fragment as falling within the near region of the terrain. The depth buffer is written as normal. At the end of this pass, the near stencil buffer marks all of the fragments that were written.

Before the terrain is rendered with the far frustum, the near stencil buffer is copied to the far stencil buffer. During rasterization of the terrain using the far frustum, the stencil test is set to pass for any fragment that does not have a value of 1 inside the far stencil buffer. At the end of rasterization, the far stencil buffer marks all the fragments that the second pass affected.

4.3.2 Secondary Objects

Using the near and far stencil buffers, an object can be rendered along with the terrain in either the range of the near frustum or the range of the far frustum by using the appropriate depth and stencil buffers. When an object in the range of the near frustum is to be rendered, the current frustum is set to the near frustum, and the near depth buffer is copied to the default framebuffer's depth and stencil buffers.

The object is then rendered as normal.

When an object in the range of the far frustum is to be rendered, the current frustum is set to the far frustum, and the far depth buffer and the far stencil buffer are copied to the default framebuffer's depth and stencil buffers. The stencil test is set to pass whenever the destination fragment is not marked by the stencil buffer as having near data written to it. If the object attempts to overwrite a pixel that the far stencil buffer marks as containing fragments written during the near pass, the fragment is discarded.

Distant objects that will always be occluded by the planet, such as stars, can be safely rendered before the terrain. The algorithm will not modify any pixels that the planet does not cover.

Chapter 5

Implementation

This chapter describes the implementation of our algorithm as the Hesperian library. We explain the motivation behind this library's software architecture, including the functional and non-functional requirements, and discuss the salient features of the exposed classes.

5.1 Requirements

Mars is the planetary body to be visualized using our algorithm. To allow visualization on multiple platforms and display environments, a number of requirements are addressed. The core terrain rendering algorithm is implemented as a library so that it can be reused in each application developed for a particular display environment. In addition, the library is implemented using cross-platform APIs, such as OpenGL, to facilitate porting to different platforms. Other APIs used include GDAL, which is useful for loading map projected images, and FreeImage [20], which is useful for loading a variety of color data formats. Both GDAL and FreeImage are cross-platform and thread safe. Also, the library is thread safe and rendering context safe. Thread safety can be ensured by using mutexes and other functionality provided by the pthread API. Rendering context safety means creating and managing rendering-context-specific data for each rendering context involved in the visualization. A rendering context is the set of all OpenGL state and data, including texture objects and vertex buffer objects allocated to the context via the OpenGL API. Multiple rendering contexts are a major requirement for virtual reality simulations where multiple screens and multiple eyes for each screen are involved.

Although Mars is the focus of the visualization, the library allows different planetary bodies to be rendered. Therefore, the library accepts a standard data format for height, color, and normal data. The planetary radius differs for each body, so the library accepts the reference radius for height data, as well as the minimum and maximum possible height values relative to the reference radius.

Because the goal is to model real planetary bodies, the library is able to use data released by NASA. Among many other types of planetary data, NASA provides height and color data along with an informative PDS label. This label contains important information about a dataset; the library is able to use the label information to correctly render this image. In addition, the library can combine overlapping datasets using our algorithm, since there could be many datasets for a particular planet and these datasets likely overlap.

The user of the library is the application, so the library provides a simple and direct interface for initializing and rendering the planet. The output screen resolution is modifiable to allow for different displays and window sizes. In addition, the capability of each display varies, so the dimensions of the projective grid is also modifiable to allow for performance adjustment.

As the visualization is running, the viewing camera's position and orientation will change. In addition, the viewing camera's frustum will vary based on the display environment. To simplify the usage of the library with respect to the variability of the viewing camera, the library infers camera attributes from the current OpenGL state instead of having the user explicitly state the viewing attributes. Therefore, the library reads the current modelview and projection matrices when rendering and update its camera information if necessary. The user may need to enable and disable datasets for rendering. To obviate the need for the visualization to be restarted, the library is able to enable and disable datasets at runtime. In addition, the display environment has a limited amount of memory, so the library loads and unload datasets as they are enabled or disabled. Because many datasets are involved, the user may also want to disable entire groups of datasets; the library provides this functionality. Finally, it is useful to scale the height values with respect to their real-world values in order to see subtle height variations. Therefore, the library allows a height scaling value to be set during the visualization.

These functional and nonfunctional requirements are summarized in Table 5.1 and Table 5.2, respectively.

F01	The library will read a standard data format for height data.
F02	The library will read standard data formats for color and normal data.
F03	The library will accept a reference radius for height data.
F04	The library will accept a minimum and a maximum height offset.
F05	The library will display data in the correct geographic area.
F06	The library will composite overlapping datasets of the same type.
F07	The library will allow the output screen size to be changed.
F08	The library will allow the projective grid size to be changed.
F09	The library will read viewing camera information directly from OpenGL.
F10	The library will allow datasets to be enabled or disabled for rendering.
F11	The library will allow groups of datasets to be enabled or disabled.
F12	The library will load data from the GPU when it is enabled.
F13	The library will unload data from the GPU when it is disabled.
F14	The library will accept a height scaling value.

Table 5.1: Functional requirements.

N01	The terrain renderer will be implemented as a library.
N06	The library will be implemented using C++.
N02	The library will be thread safe.
N03	The library will be rendering context safe.
N04	The library will use OpenGL.
N05	The library will use GLSL.
N06	The library will use GDAL for loading height data.
N07	The library will use FreeImage for loading color data.

Table 5.2: Nonfunctional requirements.

5.2 Use Cases

The use cases are shown in Figure 5.1. Before the visualization can take place, the library must be informed about each dataset to be used in the visualization. Therefore, the user is able to specify information about each dataset, including the file path, PDS label information, and texture settings. When all the datasets have been specified, the user then initializes the terrain, an operation that should happen once for the entire visualization. Next, the user initializes the graphical information for each rendering context.

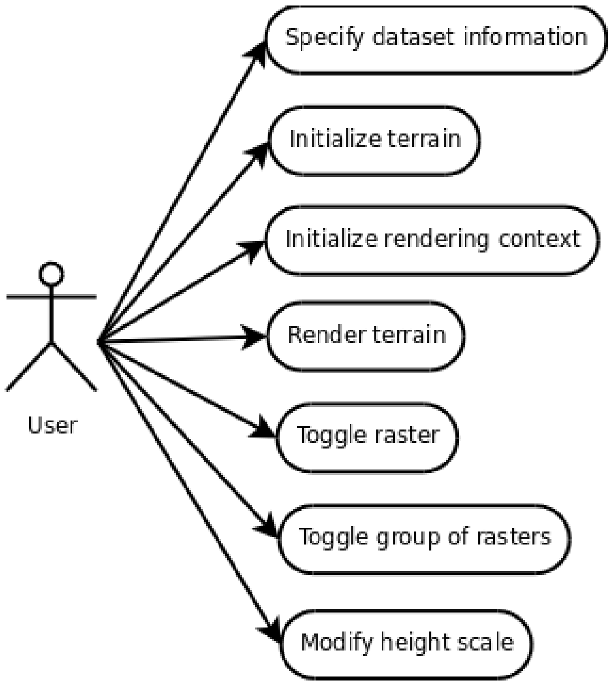


Figure 5.1: The use cases for the library.

During execution of the program, the user specifies when the terrain should be rendered during the frame. In addition, during the update step the user can toggle a raster or a group of rasters for rendering, which either enables or disables the rasters based on their current state. Also, the user may change the height scaling value during the update step.

5.3 Classes

In order to fulfill the requirements while maintaining usability and readability of code, the library requires the application programmer to interact with as few classes as possible. This section describes the external classes, which are meant to be used directly, and the internal classes, which divide the tasks required to render the terrain to improve cohesion.

5.3.1 External Classes

The external classes and their relationships are illustrated in Figure 5.2. The Terrain class serves as the primary interface between the user and the library. The initialization of Terrain is divided into two steps: initialization and rendering context initialization. The former must occur once for the entire execution of the simulation, and the latter must occur once for each rendering context involved. The initialization step requires the user to specify information about each dataset to be rendered. The user accomplishes this by requesting from Terrain a reference to an uninitialized HeightRaster or ColorRaster object. The Terrain class creates this object and returns the reference, and then the user uses the reference to modify the object with information about the dataset.

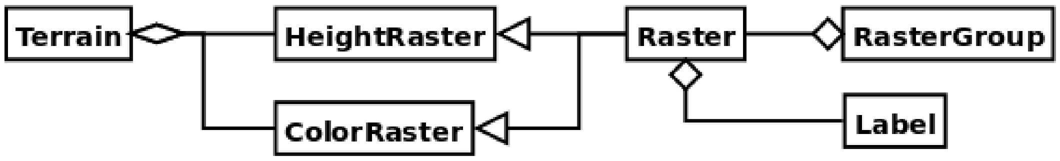


Figure 5.2: The classes with which the user interfaces.

The rendering context initialization step requires the user to make each rendering context current and call the rendering context initialization method, which signals to the terrain to create rendering-context-specific data, such as texture objects, for the currently active rendering context. In order to render the terrain, the user must set the current modelview matrix with the viewing camera's position and orientation. Then, the user must call render function of the Terrain class.

During the update step, the user can change a number of options about how the terrain is rendered. The user can change the height scaling value, which changes the relative heights of the terrain, in addition to disabling height displacement entirely. The user also has access to debug options through the Terrain class, such as enabling wireframe mode and disabling the mesh update step.

The Label structure holds PDS label information for a particular dataset. After obtaining a reference to a Raster object from Terrain, the user populates an instance of the Label structure with fields from the dataset's PDS label. This allows the application program to implement its own parser for the PDS label file, modify the PDS label (*e.g.* to scale the dataset at runtime), or supply different PDS label information as needed.

The Raster class provides an abstract interface and reusable class methods for both height and color datasets. During the initialization step, the user obtains a reference to an uninitialized Raster object from the Terrain class. When requesting a Raster, the user specifies whether a HeightRaster or ColorRaster object should be created. The Raster class allows the user to pass in a Label structure, specify the file path, and set the texture state to be used for the raster. Besides providing an abstract interface, the Raster class also contains functionality for the Terrain class to calculate uniforms required by the map projection equations on the GPU. For the purpose of enabling and disabling rasters during the visualization, the user can maintain the references to Raster objects obtained during the initialization step. The RasterGroup class further improves this functionality by allowing the user to store in an instance of RasterGroup multiple references to Raster objects. The user may then use the RasterGroup to enable or disable with a single function call each Raster object for which the RasterGroup has a reference.

5.3.2 Internal Classes

The internal classes perform functions necessary to the operation of the terrain renderer. Although the application programmer does not interact with these classes, they are important for narrowing the focus of each component of the system, thus improving cohesion. The relationships among these classes and some of the external classes are shown in Figure 5.3.

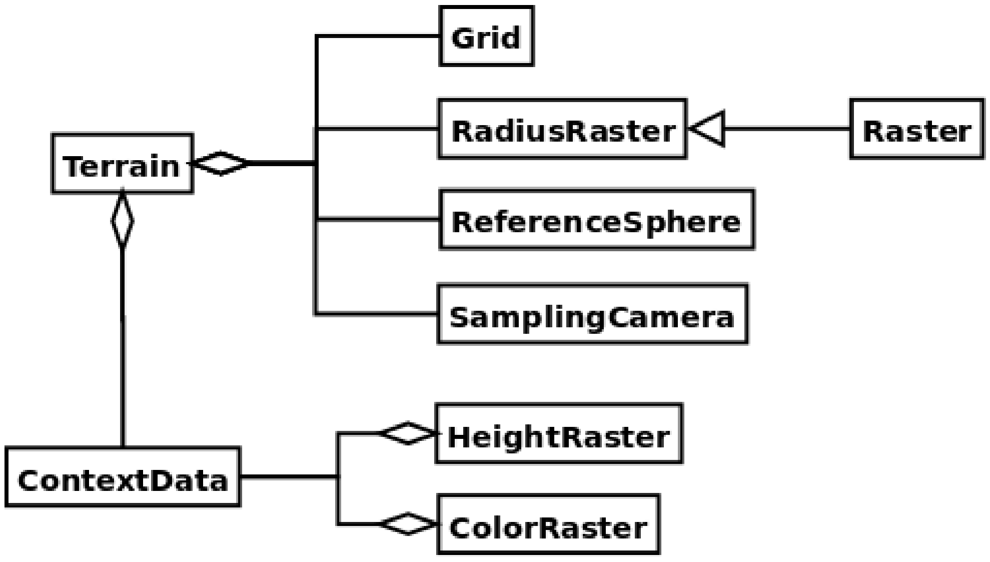


Figure 5.3: The internal classes of the library.

Although there are many graphics utility classes used within the library, the ContextData class deserves special mention. This class is used by any class that stores rendering-context-specific data, including Terrain, HeightRaster, and ColorRaster. It is a template class that allows any datatype to be specified; memory is allocated for one instance of this datatype for each rendering context. A unique identifier is assigned to each rendering context, and this identifier is used to get the instance of the datatype for a particular rendering context. Ideally, the graphics system should provide this identifier; however, in the case of a desktop application with a single rendering context, an identifier of zero can be used. Thread safety is ensured by using locks.

The Grid class is used to store a vertex buffer and index buffer for use in the rasterization step. The index buffer specifies a triangle mesh using each vertex in the grid. During initialization, it creates this grid of vertices, in which each vertex corresponds to a pixel in the position and sphere normal buffers. The RadiusRaster class derives the Raster class. It contains a global, low-resolution raster that contains the planetary radii for each point on the surface of the planet. This raster is not used for rendering; instead, it is used to efficiently query the height on the CPU.

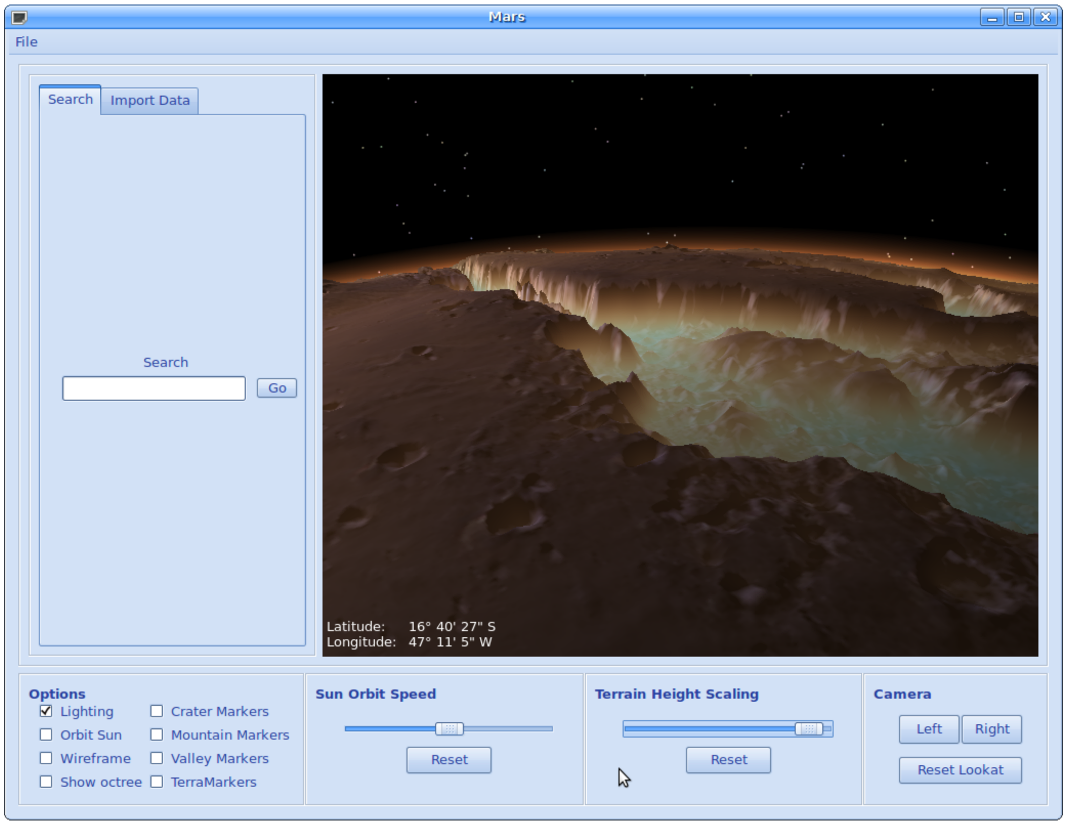
The ReferenceSphere class and SamplingCamera class are responsible for determining adequate sampling. The ReferenceSphere class calculates the reference sphere attributes for both the primary and secondary reference spheres, and the SamplingCamera class uses the ReferenceSphere class along with information about the viewing camera to generate the sampling camera. The SamplingCamera class stores a modelview and projection matrix that represent the position, orientation, and frustum of the sampling camera.

5.4 Display Environments

Using the Hesperian library along with cross-platform APIs, the Mars visualization program was developed for three display environments. The cross-platform and hardware-independent nature of the APIs, along with the architectural design of Hesperian, simplified the implementation of the visualization program for each display environment.

The first display environment is standard desktop application, as shown in Figure 5.4. This version uses Qt for window management and user interaction. Navigation is achieved by using the mouse as a virtual trackball in order to rotate about the planet.

The other two display environments are virtual reality displays that render 3D images using stereo projectors. In order to handle the management of multiple display windows, the Hydra library [30] was used. Hydra is a virtual reality library that abstracts details about the display environment. It creates the necessary rendering



Latitude: 16° 40' 27" S
Longitude: 47° 11' 5" W

Options
☑ Lighting
☐ Orbit Sun
☐ Wireframe
☐ Show octree
☐ Crater Markers
☐ Mountain Markers
☐ Valley Markers
☐ TerraMarkers

Figure 5.4: A screenshot of the desktop version of the application.

contexts for each display and assigns a unique identifier to each rendering context. The second display environment for which a visualization program was developed is a single-screen virtual reality display, as shown in Figure 5.5. This display is back projected using two projectors, one projector for each eye and uses passive stereo. The third display environment is DRIVE6, an interactive virtual environment (IVE) with six walls, which fully encloses the user. DRIVE6 is located at the Desert Research Institute, Reno, Nevada campus. Figure 5.6 shows a user interacting with the Mars visualization inside this display environment.

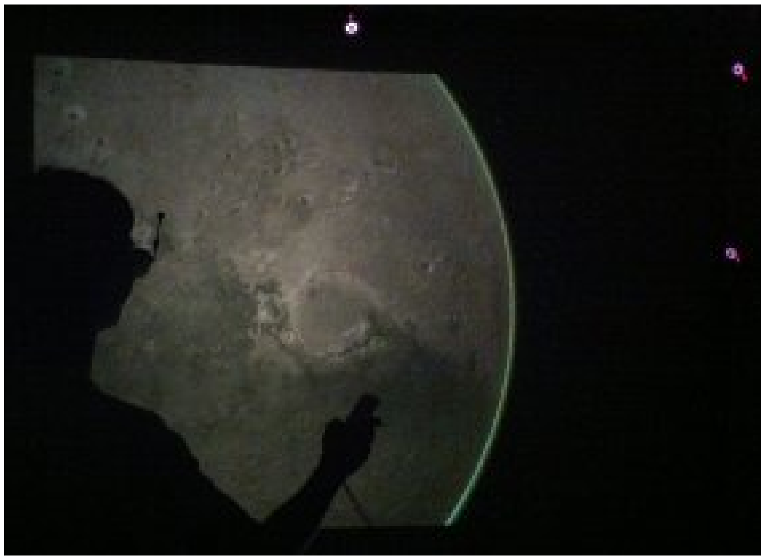


Figure 5.5: A user interacting with the single wall display.

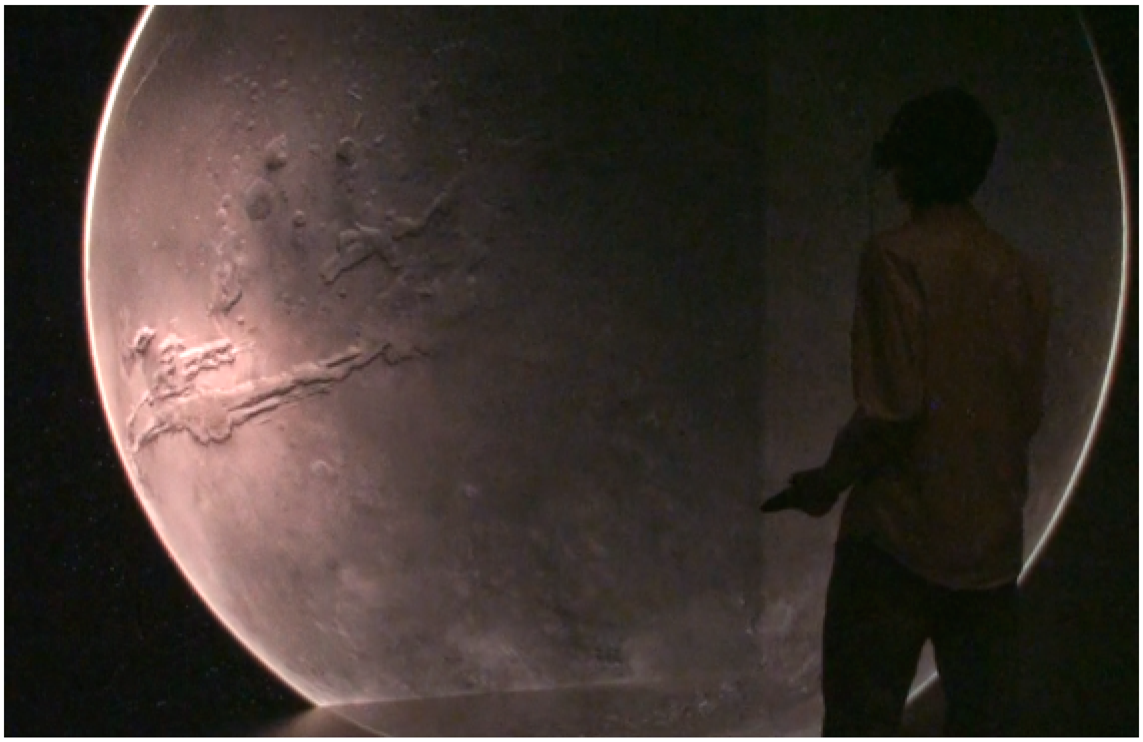


Figure 5.6: An interactive virtual environment with six walls.

Chapter 6

Results

6.1 Experimental Method

To test the execution time of the algorithm, we used a machine with an Intel Core i7 processor running at 2.8GHz and 8GB of RAM. In addition, we used an Nvidia GeForce GTX 480 graphics card with 480 shader cores clocked at 1.4MHz per core, along with 1536MB of graphics memory.

For these tests, the amount of texture data residing in GPU memory is kept constant. This is because no data streaming method is included with the algorithm, and in order to provide a fair performance estimate, we chose a static set of textures that represents a reasonable amount of terrain data to have on the GPU at one time. Although the textures do not represent all possible views of the planet to the highest fidelity, the types of data used reflect a typical usage scenario for the tested viewpoints.

The first dataset used is a 128.0MB global height map from MOLA [43]. In addition, a 63.3MB areoid height map, also from MOLA, is added to the global height map in order to correct for the oblong shape of the planet. An 8.3MB height map of Victoria Crater is used to demonstrate height composition for a focused region of the planet. This height map is a digital terrain model generated from HiRISE stereo image pairs [45]. As noted earlier, all of the height maps are converted to 32-bit floating point data.

Three color datasets are used. The first two of these datasets are tiles of the global MOC dataset [40]. Each tile is 192.0MB and covers 180° of longitude in order to provide color data for the entire planet. In addition, these tiles have been tinted to provide a photorealistic visualization of the planet. Next, a 28.4MB color map of the Victoria Crater from HiRISE is used [45]. Finally, a 96.0MB global normal map, generated from the global MOLA height map, is used for lighting. In total, 708.0MB of texture data are used for the test. Table 6.1 lists detailed information about all of the datasets used, including the map scale in kilometers per pixel, which describes the resolution of the dataset at the center of projection.

Name	**Scale (km/px)**	**Width (px)**	**Height (px)**	**Size (MB)**
MOLA Global	2.606	8192	4096	128.0
MOLA Areoid	3.705	5760	2880	63.3
Victoria Heights	0.001	1279	1694	8.3
MOC Tile 1	1.302	8192	8192	192.0
MOC Tile 2	1.302	8192	8192	192.0
Victoria Colors	0.002	3635	8192	8.3
MOLA Normals	2.606	8192	4096	96.0

Table 6.1: Information about the datasets used for timing the algorithm.

In order to understand the performance of separate parts of the algorithm, we measure just the PGM step as well as the entire algorithm. Also, we separately measure the algorithm using each of the methods for determining an approximating radius; these methods are described in Section 3.4.1. In order to see how the view affects the efficiency of the algorithm, we use three different views. The first view (Figure 6.1) encapsulates the entire planet from a distance. The second view (Figure 6.2) places the camera near the planet and at a steep angle. This view does not include high-resolution data. The final view (Figure 6.3) is similar to the second view, except that it includes high-resolution data of Victoria Crater.

Finally, the problem size is varied with respect to two variables: the grid size and the screen size. Five resolutions of grid size and screen size are used: 256×256, 256×512, 512×512, 512×1024, and 1024×1024. These resolutions were chosen to demonstrate how the algorithm responds when the number of pixels in the grid or

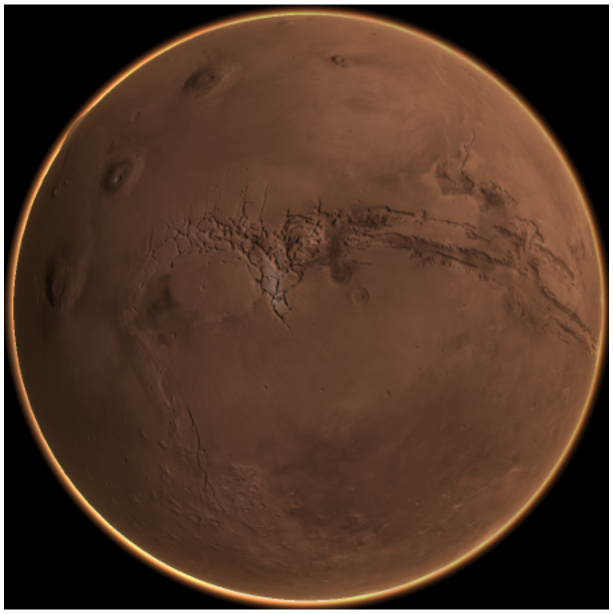


Figure 6.1: The first test case is a global view of the planet.

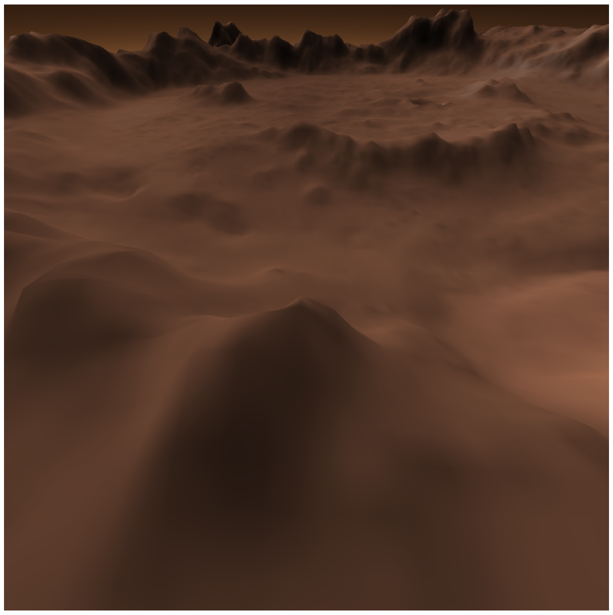


Figure 6.2: The second test case includes low-resolution data.

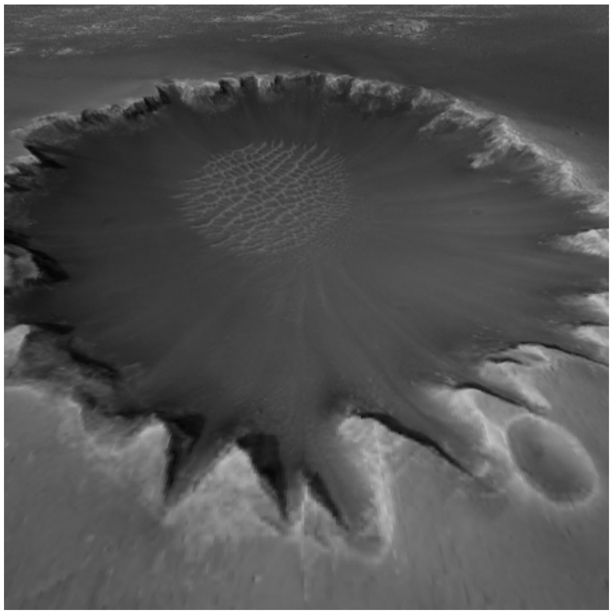


Figure 6.3: The third test case includes high-resolution data.

screen is doubled. Resolutions of 640×480, 1024×768, 1280×800, and 1920×1200 were also tested in order to show the algorithm's performance with standard screen resolutions. Also, each test case represents the average time in milliseconds over 10,000 frames.

6.2 Results and Analysis

We begin our analysis by observing how the framerate differs between the test cases. Figure 6.4 graphs the framerate in frames per second (FPS) for all three cases. This graph shows that all three cases respond similarly to changes in grid size and screen size. In addition, the graph shows that the second test case runs faster than the first test case and the third test case runs faster than the second test case. This pattern is exaggerated as both grid size and screen size approach 1024×1024. The first test case performs the slowest because it is a global view, and all of the datasets are visible. The second and third test cases are close to the terrain, meaning that texture lookups are limited to the same textures and cover a smaller area of each texture. This leads

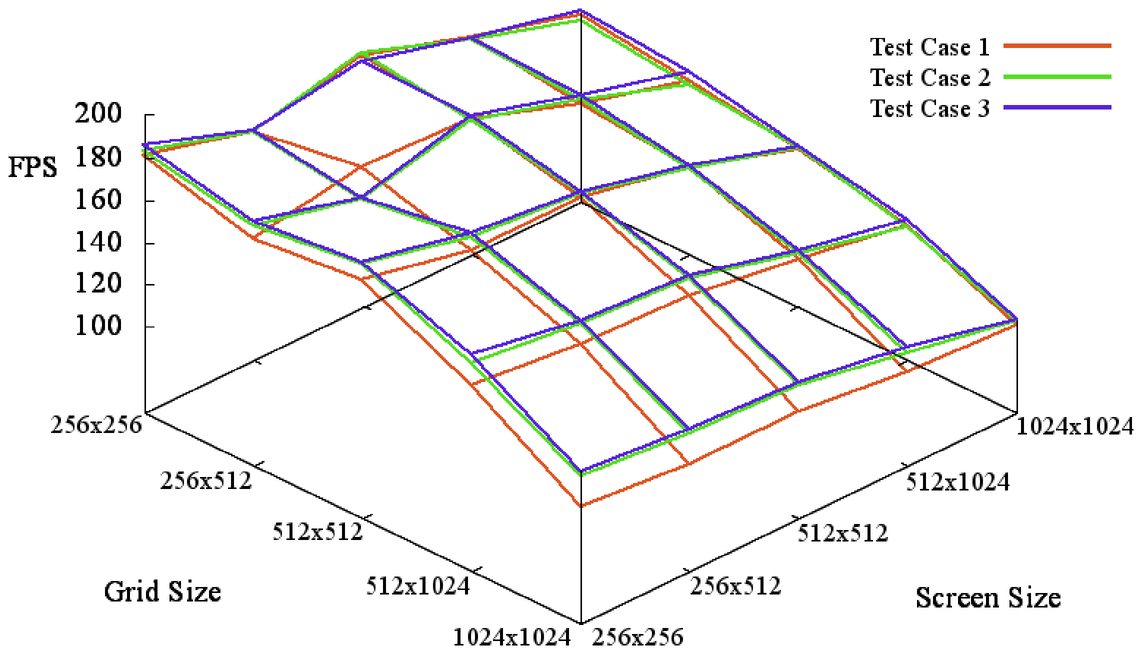


Figure 6.4: Total FPS for all three cases.

to more of a performance benefit from the caching of texture reads.

Next we observe the difference between the performance of separate parts of the algorithm. Table 6.2 shows the execution time in milliseconds of the PGM step for the first test case. Screen size increases from left to right, and grid size increases from top to bottom. In comparison, Table 6.3 shows the execution time for the entire algorithm. These data show that PGM accounts for a small portion of the execution time. With the grid size and the screen size set to 1024×1024, the PGM execution time is 17.6 times less than the execution time for the entire algorithm, or 5.6% of the overall execution time. These observations can be generalized for all three test cases since they show a similar performance degradation with respect to grid size and screen size.

The large difference in execution time between the PGM step and deferred texturing means that the most opportunity for optimization lies in deferred texturing. Our particular implementation of deferred texturing is suboptimal since a composition pass is performed for each dataset whether it is visible or not. Although the fragment shader ends early depending on whether the texture coordinates are out of bounds, this test alone is not sufficient. Instead, it is better to test for the dataset's visibility on the CPU, as Kooima [33] does. In addition, a significant amount of processing outside of PGM is devoted to building a min mipmap in order to find the smallest visible height for each frame, as described in Section 3.4.1. Table 6.4 shows the execution time for only the min mipmapping operation, which primarily depends on the size of the screen since a screen-sized buffer is used as a starting point. For a grid size and a screen size of 1024×1024, the min mipmapping operation accounts for 32.7% of the execution time.

	256×256	**256×512**	**512×512**	**512×1024**	**1024×1024**
256×256	0.16	0.15	0.15	0.16	0.16
256×512	0.18	0.19	0.19	0.19	0.20
512×512	0.23	0.23	0.24	0.23	0.24
512×1024	0.35	0.34	0.34	0.35	0.34
1024×1024	0.55	0.55	0.55	0.56	0.56

Table 6.2: Execution time in ms of the PGM step for the first test case.

	256×256	**256×512**	**512×512**	**512×1024**	**1024×1024**
256×256	5.52	5.97	5.62	6.17	6.75
256×512	6.00	5.70	5.77	6.42	7.05
512×512	5.80	6.19	6.21	6.63	7.45
512×1024	6.78	7.03	7.15	7.55	8.10
1024×1024	8.64	9.01	9.05	9.55	9.85

Table 6.3: Execution time in ms of the entire algorithm for the first test case.

	256×256	**256×512**	**512×512**	**512×1024**	**1024×1024**
256×256	2.96	3.42	3.01	3.46	3.20
256×512	3.43	3.14	3.13	3.69	3.17
512×512	3.23	3.60	3.57	3.66	3.19
512×1024	3.82	3.92	3.80	3.67	3.08
1024×1024	3.78	4.01	3.83	3.85	3.22

Table 6.4: Execution time in ms of min mipmapping for the first test case.

The PGM step is unaffected by the dimensions of the screen because of the nature of the PGM algorithm. This observation is shown in Figure 6.5, where the execution time shows small variations with increasing screen size but large variations with increasing grid size. However, both the grid size and the screen size affect the overall performance of the algorithm, as shown in Figure 6.6. This graph shows that although deferred texturing operates on screen-sized buffers, an increase in grid size affects the overall execution time of the algorithm more than an increase in screen size. This is because the rasterization step requires the entire projected grid to be rendered as a triangle mesh at once, thus testing the triangle throughput of the GPU.

Finally, Table 6.5 shows the performance of the algorithm in FPS for standard screen resolutions. In each of these cases, the grid size is equal to the screen size. The algorithm performs reasonably well for these resolutions.

Resolution	**Test Case 1**	**Test Case 2**	**Test Case 3**
640×480	154.9	159.4	159.0
1024×768	113.6	120.4	120.7
1280×800	101.2	105.8	106.5
1920×1200	64.5	70.2	72.1

Table 6.5: Performance of the entire algorithm in FPS for all three test cases.

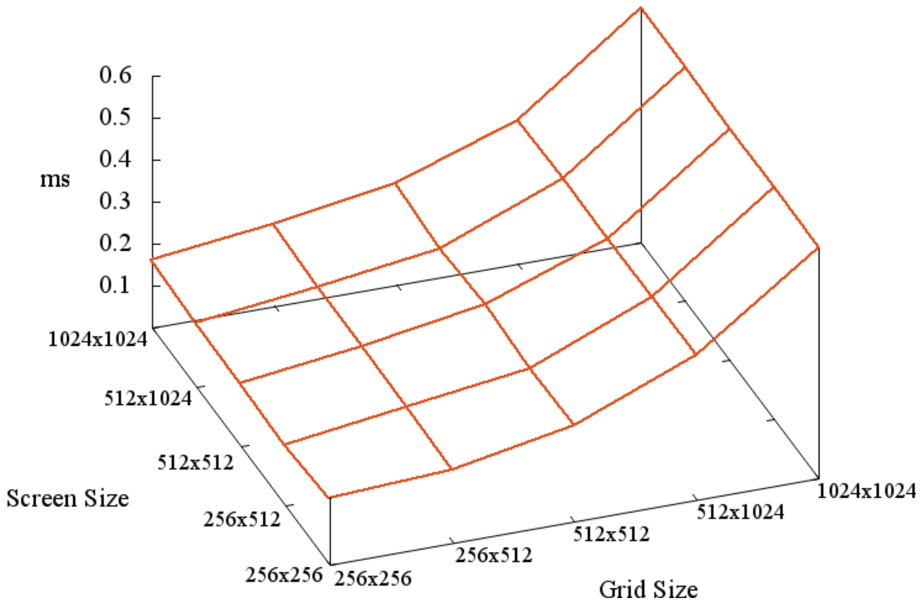


Figure 6.5: The execution time in ms of the PGM step for the first test case.

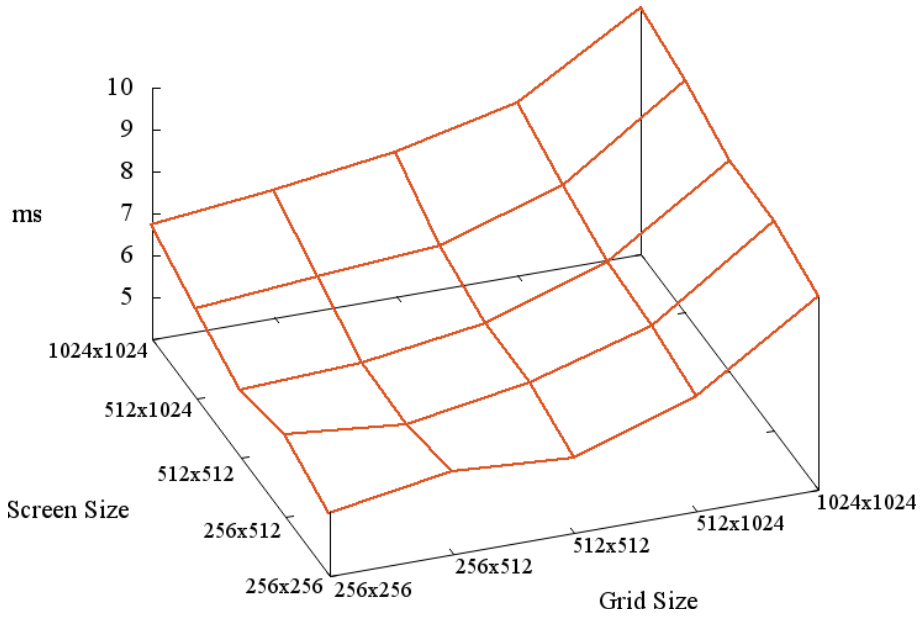


Figure 6.6: The execution time in ms of the entire algorithm for the first test case.

6.3 Visual Results

This section describes the benefits and drawbacks with respect to image quality. Although a grid size that is equal to or less than the screen size leads to reasonable framerates, artifacts result from undersampling the terrain in areas of high frequency. Figure 6.7 displays a close-up of a sharp peak, exaggerated five times in order to clearly show the effect. For low-frequency areas of the terrain and for terrain without height scaling, these artifacts are less apparent. This problem can be mitigated by using a grid size larger than the screen size, but at the cost of more processing time.

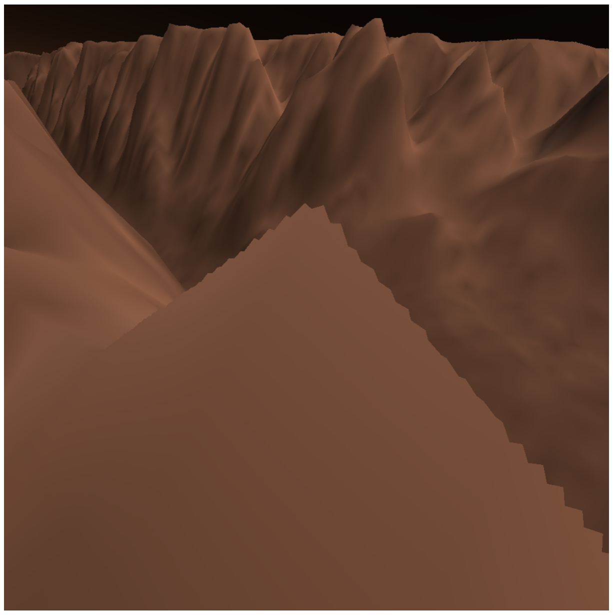


Figure 6.7: Artifacts result from undersampling high-frequency terrain.

Figure 6.8 demonstrates height and color maps using the equirectangular projection. The datasets used show the Victoria Crater and were obtained from HiRISE [45]. The left image shows the crater from above. The right image shows the crater from inside; note that without correct occlusion testing with the near and far regions of the terrain, the far regions would be visible in front of the crater wall. Also, Figure 6.9 demonstrates the rendering of a HiRISE dataset that uses a polar stereographic projection at Mars's north pole.

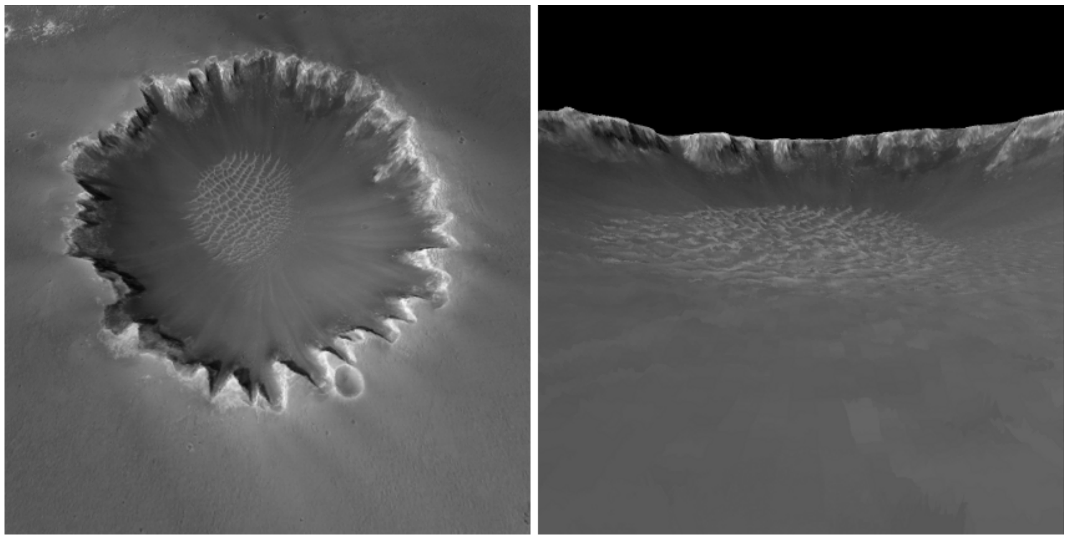


Figure 6.8: Victoria Crater from above and inside the crater.

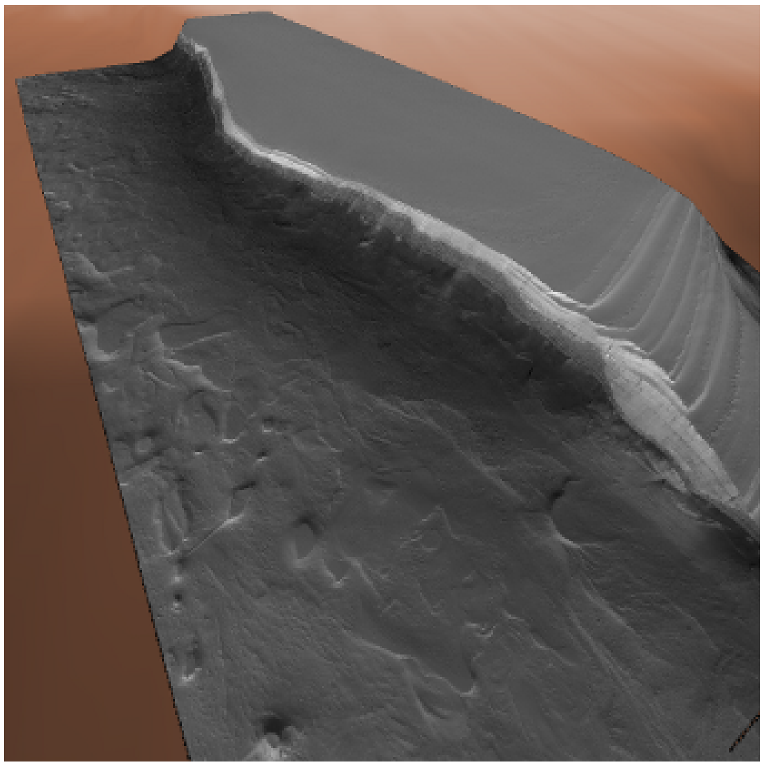


Figure 6.9: A HiRISE dataset using a polar stereographic projection.

Finally, Figure 6.10 demonstrates the difference between using a global normal map and a polar normal map. In the image on the left, a normal map generated from the global MOLA height data is used. The lighting is blurred around the cliffs and there is a clear pattern of lines extending in the direction of the pole. In the image on the right, a normal map generated from the north polar height map is used, which resolves the lighting artifacts in the first image.

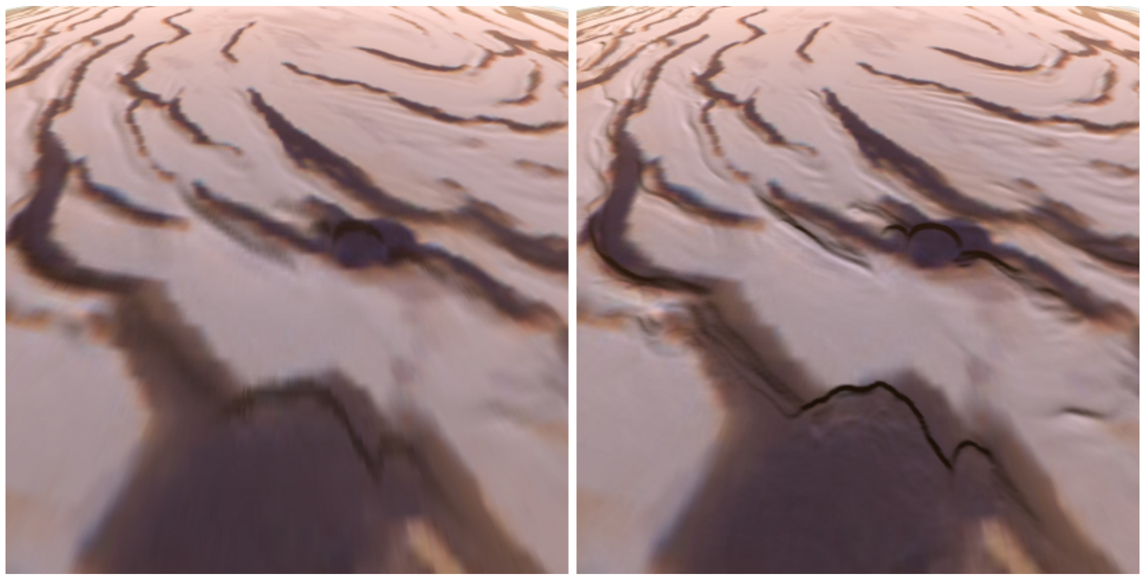


Figure 6.10: Using normals in a polar stereographic projection reduces artifacts.

Chapter 7

Conclusions and Future Work

The advantage of PGM is that it operates on the GPU, generating the terrain mesh in GPU memory and removing the need for terrain data to move between the CPU and the GPU before rendering. With the parallelism of modern GPUs, PGM is a relatively efficient operation. Although PGM presents the challenge of adequately sampling the terrain, the use of reference spheres and an appropriately positioned and oriented sampling camera can accomplish this sampling. The primary difficulty presented by PGM is the brute-force rasterization of the entire projected grid during each frame. In addition, the most effective but costly method of determining the primary reference sphere radius accounts for more than a third of the processing time. Performance throttling is possible with PGM by changing either the grid size or the screen size; however, depending on the frequency of the terrain function, a grid size larger than the screen size may be necessary to mitigate undersampling.

We have applied the PGM technique to planetary rendering. In formulating a spherical version of PGM, we presented a method for obtaining an adequate sampling of a spherical reference surface. This includes the reduction of ray-sphere intersection to the two-dimensional problem of ray-circle intersection, which leads to simple and direct GPU code to project each grid point onto the reference sphere. Also, we explained the methods by which reference spheres for ray casting are selected, and described the algorithm for calculating a sampling camera that generates all of the necessary eye rays.

In addition to spherical PGM, we gave a method for transforming geographic coordinates to texture coordinates for two commonly used map projections. Also, we provided a technique for handling polar projected normal maps uniformly with equirectangular projected normal maps. Finally, we described how secondary objects could be correctly occluded with the terrain, even when using two passes to render the near and far portions of the terrain.

Based on the analysis of our results, there are three ways in which the performance of our algorithm could be improved. First, the entire projected grid is rasterized as a triangle mesh every frame; generating a hierarchical triangle mesh based on the grid points normals and height values may prove beneficial if it can be efficiently implemented on the GPU. Second, the min mipmapping approach for selecting the radius of the primary reference sphere is effective but slow since the mipmapping algorithm uses exponentially fewer shader cores as it approaches the lower levels of the mipmap hierarchy. This approach could be optimized using CUDA [48] to manage the usage of shader cores and perform the reduction of the last mipmap levels on the CPU, or an alternative approach for finding the minimum visible height could be developed. Finally, our implementation of deferred texturing does not organize the datasets into a spatial data structure; using a structure such as a bounding volume hierarchy and testing for visibility, entire composition passes could be eliminated.

There is also an opportunity for performance gain by reusing the processing from previous frames since consecutive frames exhibit coherence in the geographic areas visualized. This optimization is currently precluded by the generation of ray-sphere intersections relative to the eye, which we do in order to mitigate floating point precision issues. Similarly, splitting the work of the algorithm across multiple frames in a process called amortization would also lead to an improvement in framerate and give another variable for performance throttling.

Our approach focuses on PGM. However, we do not provide any method for streaming data from disk to RAM and from RAM to GPU memory. Such a method would also include mipmapping various levels of the height and color textures used in the visualization. A drawback of the GPU-focused approach to terrain rendering is that the generated mesh cannot be used for tasks other than rendering, such as collision detection. In order to efficiently perform collision detection on the GPU, multiple collision checks would have to be batched and processed together on the GPU. Deformation is another operation that should be implemented on the GPU in order to avoid transferring the mesh from the GPU to the CPU.

Just as we have extended PGM to use a sphere instead of a plane as the underlying surface, the algorithm could be further extended to support nonspherical bodies. This could be accomplished by projecting the grid onto an ellipsoid or collection of ellipsoids that approximate a nonspherical body. Also, the precision of floating point numbers remains a problem when it comes to sampling textures since the projection equations require the use of trigonometric functions. The effect of this problem could be reduced by using a look-up table or reformulating the projection equations.

Finally, as mentioned earlier, our method of PGM suffers from artifacts due to undersampling. For the purpose of rigorous application to scientific visualization, a method for measuring the error of the generated terrain mesh is required. In addition, the use of an error metric could be used to dynamically adjust the grid size and distribution of grid points to provide a more accurate sampling of the terrain.

Bibliography

[1] E. Angel. *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*, pages 289–304, 492–495. Addison Wesley, 5th edition, April 2008.

[2] A. Asirvatham and H. Hoppe. Terrain rendering using GPU-based geometry clipmaps. In *GPU Gems 2*, pages 27–46. Addison-Wesley Professional, March 2005.

[3] W. E. Brandstetter. Multi-resolution deformation in out-of-core terrain rendering. Master's thesis, Department of Computer Science and Engineering, University of Nevada, Reno, December 2007.

[4] W. E. Brandstetter, J. D. Mahsman, C. J. White, S. M. Dascalu, and F. C. Harris. Multi-resolution deformation in out-of-core terrain rendering. In *Proceedings of ISCA's 23rd International Conference on Computer Applications in Industry and Engineering, (CAINE '10)*, November 2010.

[5] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Planet-sized batched dynamic adaptive meshes (P-BDAM). In *Proceedings of the 14th IEEE Visualization 2003*, pages 147–154. IEEE Computer Society, 2003.

[6] P. Cignoni, F. Ganovelli, E. Gobetti, F. Marton, F. Ponchio, and R. Scopigno. BDAM—Batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum*, 22(3):505–514, September 2003.

[7] M. Clasen and H. Hege. Terrain rendering using spherical clipmaps. In *EuroVis06: Joint Eurographics - IEEE VGTC Symposium on Visualization*, pages 91–98. Eurographics Association, 2006.

[8] D. Cohen-Or and Y. Levanoni. Temporal continuity of levels of detail in Delaunay triangulated terrain. In *Proceedings of the 7th conference on Visualization 1996*, pages 37–42. IEEE Computer Society, October 1996.

[9] R. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231. ACM, 1984.

[10] R. Cook, L. Carpenter, and E. Catmull. The Reyes image rendering architecture. *ACM SIGGRAPH Computer Graphics*, 21(4):95–102, July 1987.

[11] C. Dachsbacher and M. Stamminger. Rendering procedural terrain by geometry image warping. In *Eurographics Symposium on Geometry Processing*, pages 138–145. Eurographics Association, 2004.

[12] W. De Boer. Fast terrain rendering using geometrical mipmapping. http://www.flipcode.com/archives/article_geomipmaps.pdf (Accessed July 26, 2010).

[13] C. Dick, J. Krüger, and R. Westermann. GPU ray-casting for scalable terrain rendering. In *Proceedings of Eurographics 2009–Areas Papers*, pages 43–50. Eurographics Association, 2009.

[14] C. Dick, J. Schneider, and R. Westermann. Efficient geometry compression for GPU-based decoding in realtime terrain rendering. *Computer Graphics Forum*, 28(1):67–83, 2009.

[15] W. Donnelly. Per-pixel displacement mapping with distance functions. In *GPU Gems 2*, pages 123–136. Addison-Wesley Professional, March 2005.

[16] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 81–88. IEEE Computer Society Press, 1997.

[17] J. Dummer. Cone step mapping: An iterative ray-heightfield intersection algorithm. http://www.lonesock.net/files/ConeStepMapping.pdf (Accessed July 26, 2010).

[18] E. Eliason. Hirise catalog. http://hirise.lpl.arizona.edu/PDS/CATALOG/DSMAP.CAT (Accessed July 21, 2010).

[19] E. Fourquet, W. Cowan, and S. Mann. Geometric displacement on plane and sphere. In *Proceedings of graphics interface 2008*, pages 193–202. Canadian Information Processing Society, 2008.

[20] FreeImage. http://freeimage.sourceforge.net/ (Accessed October 27, 2010).

[21] M. Garland and P. Heckbert. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, Computer Science Department, Carnegie Mellon University, September 1995.

[22] Gazeteer of Planetary Nomenclature. Mars coordinate systems. http://planetarynames.wr.usgs.gov/Page/MARS/system (Accessed July 21, 2010).

[23] GDAL. http://www.gdal.org (Accessed July 21, 2010).

[24] Google Earth. http://earth.google.com (Accessed April 21, 2010).

[25] P. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. In *Multiresolution surface modeling (SIGGRAPH '97 Course notes 25)*. ACM, 1997.

[26] D. Hinsinger, F. Neyret, and M. Cani. Interactive animation of ocean waves. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 161–166. ACM, 2002.

[27] H. Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 189–198. ACM Press/Addison-Wesley Publishing Co., 1997.

[28] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 35–42. IEEE Computer Society Press, 1998.

[29] L. Hu, P. Sander, and H. Hoppe. Parallel view-dependent level-of-detail control. *IEEE Transactions on Visualization and Computer Graphics*, 16(5):718–728, 2010.

[30] Hydra. `http://www.cse.unr.edu/hpcvis/hydra/` (Accessed August 26, 2010).

[31] C. Johanson. Real-time water rendering. Master's thesis, Department of Computer Science, Lund University, March 2004.

[32] R. Knippers. Geometric aspects of mapping. `http://www.kartografie.nl/geometrics` (Accessed April 21, 2010), 2009.

[33] R. Kooima. *Planetary-scale Terrain Composition.* PhD dissertation, Department of Computer Science, University of Illinois at Chicago, 2008.

[34] R. Kooima, J. Leigh, A. Johnson, D. Roberts, M. SubbaRao, and T. DeFanti. Planetary-scale terrain composition. *IEEE Transactions on Visualization and Computer Graphics*, 15(5):719–733, 2009.

[35] T. Lauritsen and S. Nielsen. Rendering very large, very detailed terrains. `http://www.terrain.dk/terrain.pdf` (Accessed July 26, 2010).

[36] 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. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118. ACM, 1996.

[37] Y. Livny, N. Sokolovsky, T. Grinshpoun, and J. El-Sana. A GPU persistent grid mapping for terrain rendering. *The Visual Computer*, 24(2):139–153, 2008.

[38] F. Löffler, S. Rybacki, and H. Schumann. Error-bounded GPU-supported terrain visualisation. In *WSCG'2009: Proceedings of the 17th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision–Communications Papers*, pages 47–54. University of West Bohemia, 2009.

[39] F. Losasso and H. Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 769–776. ACM, 2004.

[40] Malin Space Science Systems. Mars Orbiter Camera Geodesy Campaign Mosaic. `http://www.msss.com/mgcwg/mgm/` (Accessed September 8, 2010).

[41] NASA. Landsat Image Mosaic of Antarctica Faces of Antarctica. `http://lima.nasa.gov` (Accessed April 21, 2010).

[42] NASA. PDS: Planetary Data System. http://pds.nasa.gov (Accessed July 21, 2010).

[43] NASA Goddard Space Flight Center. The Mars Orbiter Laster Altimiter. http://mola.gsfc.nasa.gov (Accessed April 21, 2010).

[44] NASA Jet Propulsion Laboratory. Shuttle Radar Topograpy Mission. http://www2.jpl.nasa.gov/srtm (Accessed April 21, 2010).

[45] NASA, JPL, and University of Arizona. HiRISE: High Resolution Imaging Science Experiment. http://hirise.lpl.arizona.edu (Accessed July 21, 2010).

[46] NASA, JPL, and University of Arizona. Victoria Crater at Meridiani Planum (TRA_000873_1780). http://hirise.lpl.arizona.edu/TRA_000873_1780 (Accessed July 21, 2010).

[47] NSIDC National Snow and Ice Data Center. MODIS Mosaic of Antarctica (MOA) Image Map. http://nsidc.org/data/nsidc-0280.html (Accessed April 21, 2010).

[48] Nvidia. CUDA. http://www.nvidia.com/object/cuda_home_new.html (Accessed September 13, 2010).

[49] M. Oliveira and F. Policarpo. An efficient representation for surface details. Technical Report RP-351, Instituto de Informatica UFRGS, January 2005.

[50] S. O'Neil. A real-time procedural universe, part three: Matters of scale. http://www.gamasutra.com/features/20020712/oneil01.htm (Accessed August 25, 2010).

[51] S. O'Neil. Accurate atmospheric scattering. In *GPU Gems 2*, pages 253–268. Addison-Wesley Professional, March 2005.

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

[53] R. Pajarola and E. Gobbetti. Survey of semi-regular multiresolution models for interactive terrain rendering. *The Visual Computer*, 23(8):583–605, 2007.

[54] F. Policarpo, F. Fonseca, and C. Games. Deferred Shading Tutorial. http://artis.inrialpes.fr/Membres/Olivier.Hoel/deferred_shading/Deferred_Shading_Tutorial.pdf (Accessed October 27, 2010).

[55] F. Policarpo and M. Oliveira. Relaxed cone stepping for relief mapping. In *GPU Gems 3*, pages 409–428. Addison-Wesley Professional, August 2007.

[56] A. Pomeranz. ROAM using surface triangle clusters (RUSTIC). Master's thesis, Department of Computer Science, University of California, Davis, 1998.

[57] J. Schneider, T. Boldte, and R. Westermann. Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In *Proceedings of Vision, Modeling and Visualization 2006*, pages 145–152. IOS Press, November 2006.

[58] P. Seidelmann, B. Archinal, M. Ahearn, A. Conrad, G. Consolmagno, D. Hestroffer, J. Hilton, G. Krasinsky, G. Neumann, J. Oberst, P. Stooke, E. Tedesco, D. Tholen, P. Thomas, and I. Williams. Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006. *Celestial Mechanics and Dynamical Astronomy*, 98(3):155–180, July 2007.

[59] L. Spector. The Math Page. `http://www.themathpage.com/atrig/law-of-sines.htm` (Accessed August 17, 2010).

[60] L. Szirmay-Kalos and T. Umenhoffer. Displacement mapping on the GPU–State of the art. *Computer Graphics Forum*, 27(6):1567–1592, September 2008.

[61] C. Tanner, C. Migdal, and M. Jones. The clipmap: A virtual mipmap. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158. ACM, 1998.

[62] N. Tatarchuk. Dynamic parallax occlusion mapping with approximate soft shadows. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, page 69. ACM, 2006.

[63] A. Tevs, I. Ihrke, and H.-P. Seidel. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 183–190. ACM, 2008.

[64] The Internet Encyclopedia of Science. Planetocentric and planetographic coordinates. `http://www.daviddarling.info/encyclopedia/P/planetocentric_coordinates.html` (Accessed July 21, 2010).

[65] T. Ulrich. Rendering massive terrains using chunked level of detail control. `http://tulrich.com/geekstuff/chunklod.html` (Accessed October 27, 2010).

[66] University of Arizona. Mars Global Data Sets. `http://www.mars.asu.edu/data` (Accessed July 21, 2010).

[67] L. Van Zijl. Computer Graphics. `http://www.cs.sun.ac.za/~lvzijl/courses/rw778/grafika/OpenGLtuts/Big/graphicsnotes005.html` (Accessed August 16, 2010).

[68] B. Von Herzen and A. H. Barr. Accurate triangulations of deformed, intersecting surfaces. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 103–110. ACM, 1987.