Projective Grid Mapping for Planetary Terrain

Joseph Mahsman[1] Cody White[1] Daniel Coming[2] Frederick C. Harris[1]

[1]University of Nevada, Reno
[2]Desert Research Institute, Reno, Nevada

Abstract. Visualizing planetary terrain presents unique challenges not found in traditional terrain visualization: displaying features that rise from behind the horizon on a curved surface; rendering near and far surfaces with large differences in scale; and using appropriate map projections, especially near the poles. Projective Grid Mapping (PGM) is a hybrid terrain rendering technique that combines ray casting and rasterization to create a view-dependent mesh in GPU memory. We formulate a version of PGM for planetary terrain that simplifies ray-sphere intersection to two dimensions and ray casts the area behind the horizon. We minimize the number of rays that do not contribute to the final image by creating a camera that samples only the area of the planet that could affect the final image. The GPU-resident mesh obviates host-to-device communication of vertices and allows for dynamic operations, such as compositing overlapping datasets. We adapt two map projections to the GPU for equatorial and polar data to reduce distortion. Our method achieves realtime framerates for a full-planet visualization of Mars.

1 Introduction

There are three problems of planetary terrain that this work addresses. The curved body of the planet means that for a viewer on the surface there is a horizon that marks the boundary between what can be seen and what cannot be seen. The terrain algorithm must ensure that the area behind the horizon is adequately sampled so that any mountains that affect the image are sampled. The second problem is the issue of the large scale of the planet. This affects the z-buffering algorithm which only maintains a limited precision buffer for determining surface depths. The final problem is using datasets in appropriate map projections for the area depicted. This problem commonly demonstrates itself near the poles of planetary terrain renderers. Distortion is visible since the underlying map projection used is meant for the equatorial region, and yet the square mipmap filter is being applied to the distorted polar regions of the mipmap.

We choose to adapt a planar terrain algorithm to the sphere to enable planetary rendering. Projective Grid Mapping (PGM) is chosen since it allows the creation of the terrain mesh entirely in GPU memory. This obviates the need for vertex transfer between the CPU and the GPU. PGM also generates the mesh in a view-dependent manner with gradual detail transition moving further away from the viewer.

The GPU-resident mesh representation allows us to apply dynamic, massively parallel operations to the terrain mesh at runtime. In the case of planetary data, usually the visualization involves multiple data maps (heights and colors) that might overlap. Terrain composition as described by Kooima *et al.* [1] allows for these overlapping datasets to be combined at runtime. Using deferred texturing and PGM, we implement a full planetary terrain renderer.

In order to simplify ray-sphere intersection and to create a conceptual framework through which to solve sampling problems, described later, we reduce ray-sphere intersection to two dimensions as a function of a ray's angle to the line connecting the viewpoint and the sphere's center (Section 3).

Before PGM can be applied to the sphere we must avoid sampling issues. This is done by selecting reference spheres for ray casting, and generating a sampling camera (Section 4). The entities are used in ray casting the sphere.

The purpose of reference spheres is to avoid sampling issues when ray casting. The sampling camera minimizes the number of rays that don't contribute either because they miss or they sample the area of the planet that does not affect the final image (Section 5).

Finally, the ray casting step uses these entities (Section 3). The goal is to generate positions relative to the camera to avoid artifacts due to 32-bit floating point imprecision. The output of PGM are two buffers, sized to the projective grid; the positions and sphere normals. Positions are used as points on the sphere to be displaced later. Sphere normals are used to determine a geographic location.

These buffers are fed into the height compositor as done by Kooima *et al.* [1]. The mesh is then rasterized by displacing each grid point along its composited height value. Colors are then composited in screen space. In order to read the data maps, map projections must be implemented on the GPU. We formulate versions of these equations for the GPU (Section 7). Finally, lighting is applied.

We demonstrate the performance of this algorithm and identify some of its weaknesses in a simulation of a full-planet of Mars (Section 8).

2 Related Work

Terrain algorithms can be classified based on the underlying surface used for displacement. Extensive research has been performed for plane-based algorithms. Real-time Optimally Adapting Meshes (ROAM) by Duchaineau *et al.* [2] is an effective CPU-based algorithm that progressively refines the terrain mesh for the current view. Losasso and Hoppe [3] present geometry clipmapping, which is capable of handling large amounts of static terrain data. Asirvatham and Hoppe [4] extend geometry clipmaps to use programmable shaders, making the approach one of the first terrain algorithms to take advantage of modern GPU capabilities. Dick *et al.* [5] implement ray casting for terrain and find success with large amounts of data.

There are few terrain rendering algorithms that address the issue of rendering on a sphere or planetary body. Cignoni *et al.* [6] describe Planetary Batched Dynamic Adaptive Meshes (P-BDAM). 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.

Clasen and Hege [7] update the geometry clipmapping approach of Losasso and Hoppe [3] for planetary terrain. Although successful, their approach generates the mesh for the visible part of the user's current hemisphere entirely, and does not perform any frustum culling. Therefore, no matter where the user is looking, there is always wasted triangles being rendered. In addition, this approach requires the terrain to be static.

Kooima *et al.* [1] present a method of planetary terrain rendering that builds a terrain mesh using the CPU and the GPU. The algorithm creates a mesh with vertices on the sphere, and later displaces these vertices based on composited heights. First, coarse-grained refinement is performed on the CPU to determine view-dependent levels of detail within the mesh. Next, the coarse mesh is refined on the GPU to create the final terrain. The mesh exists on the GPU as raster data, allowing for various height maps to be composited and displace the mesh.

To create a planetary algorithm that operates entirely on the GPU, we adapt the projected grid approach or PGM for planetary rendering. PGM is a technique for approximating a surface displaced from a plane. The technique was first applied as a method for visualization of waves by Hinsinger *et al.* [8] and later extended by Johanson [9]. Bruneton *et al.* [10] currently uses a projected grid for ocean wave animation.

PGM was first applied to terrain by Livny *et al.* [11] and later extended to make use of screen space error metrics to guide sampling by Löffler *et al.* [12]. Schneider *et al.* [13] use PGM for planetary terrain synthesis, however they do not handle terrain beyond the horizon, instead electing to obscure the horizon with atmosphere and fog effects.

PGM begins with a regular grid of points stretched across the viewport. Each point is projected onto the terrain along its corresponding eye ray. The terrain intersection point of each projected grid point corresponds to a location within the height map. Therefore at each projected grid point, the height map is queried and the point is displaced vertically along the base plane normal. The projected and displaced grid is now interpreted as a triangle mesh and rasterized.

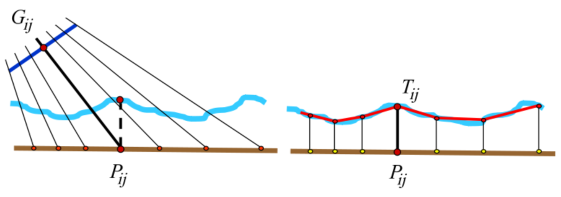

Fig. 1. The grid projected onto the terrain base plane.

There are some problem with this. The viewing camera may miss areas of the terrain below the camera which could include mountains that should intersect the viewing camera. This is solved by creating a special sampling camera. Another problem is that the sampling rate of the projected grid becomes much lower than that of the height map as the distance from the viewpoint increases; although this leads to a positive, gradual reduction of detail, the low sampling leads to missing details and the appearance of wavy mesh. This can be addressed by either increasing the number of samples of the projective grid or using filtered versions of the height map as the distance increases to reduce the sampling frequency of the terrain function.

3 Projective Grid Mapping for Planetary Terrain

For each grid point, we need to generate a ray to ray cast the reference spheres. Since we reduce the intersection to two-dimensions, we must also generate the proper coordinate system transformation for the generated rays. Finally, we must determine the position of the ray-circle intersection. This position must be calculated in eye space so as to mitigate floating point precision issues that would arise from measuring the position relative to world space.

The output of PGM consists of two buffers, where each pixel represents data for a grid point. The first buffer is the positions buffer that holds the ray-sphere intersection point. The second buffer contains the normals to the sphere at the intersection points.

Eye Rays. PGM is carried out by binding the output buffers to the current framebuffer and rendering a fullscreen quad. Each fragment represents a grid point. Thus, we can generate the eye rays for each screen fragment in parallel. The sampling camera matrix and the inverse of the sampling projection matrix are uploaded to the GPU and a fullscreen quad is rendered to start this process. The derivation of these matrices is described in Section 5.

In order to find the eye ray associated with the current fragment, we must first transform the current fragment coordinate into clip space. We can then move into sampling camera coordinates with the use of the inverse sampling camera projection matrix uploaded earlier. A perspective divide is then performed to finish the transformation. Since the origin of the 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. Therefore, we can normalize the position and get the eye ray for the grid point.

Two-Dimensional Reduction. Inspired by the work of Fourquet *et al.* [14] in displacement mapping a spherical underlying surface, we simplify ray-sphere intersection in three dimensions to ray-circle intersection in two dimensions. This approach not only simplifies the GPU implementation, but also provides a mathematical framework to solve the sampling issues (Section 4 and Section 5).

We derive the trigonometric equation that maps the elevation ω of an eye ray to the elevation γ of the sphere normal at the ray-sphere intersection point, as shown in Figure 2. 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 $\widehat{s}$ at the ray-sphere intersection point I and the antinadir $-\widehat{n}$. All vectors are assumed to be unit vectors unless otherwise noted.

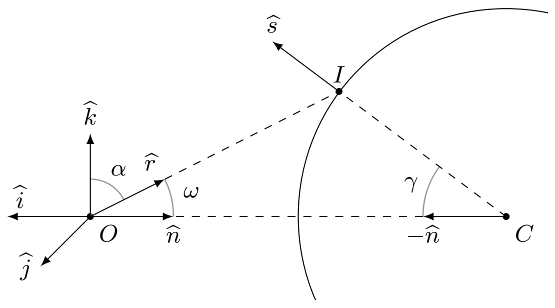

Fig. 2. The calculation of the ray angle.

In order to convert an eye ray to its ray angle ω, we must calculate the local coordinate system. This calculation begins with determining 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 then produces the up vector.

The three basis vectors of the coordinate system are $\widehat{i}$, $\widehat{j}$, and $\widehat{k}$, corresponding to the x-, y-, and z-axes. The nadir $\widehat{n}$ is taken to be the first basis vector $\widehat{i}$. The cross product of $\widehat{i}$ and $\widehat{r}$ produces the right vector $\widehat{j}$. Finally, the cross product of $\widehat{j}$ and $\widehat{i}$ produces the up vector $\widehat{k}$.

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 2 shows the angle ω between $\widehat{r}$ and $\widehat{n}$, as well as the angle α between $\widehat{r}$ and $\widehat{k}$. Then the equation

$$\widehat{r} \cdot \widehat{k} = \cos\alpha \tag{1}$$

relates $\widehat{r}$ and $\widehat{k}$ to the angle α between them. Using the knowledge that $\widehat{k}$ is perpendicular to $\widehat{n}$, we note that $\omega = \pi/2 - \alpha$. The trigonometric identity $\cos\alpha = \sin(\pi/2 - \alpha)$ implies that $\cos\alpha = \sin\omega$. Substituting this result into Equation 1, we get

$$\omega = \operatorname{asin}(\widehat{r} \cdot \widehat{k}). \tag{2}$$

We can now calculate the normal angle as a function of the ray angle. Figure 3 illustrates the situation with a ray cast against a sphere. Points I_1 and I_2

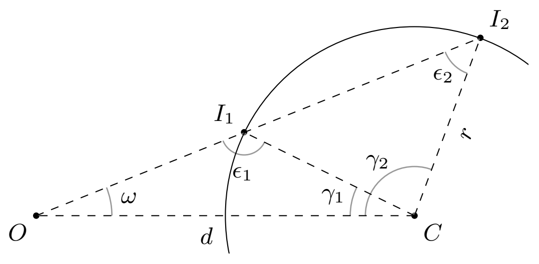

Fig. 3. Ambiguous solutions using the law of sines.

represent the first and second intersections of the ray with the sphere. The angles γ_1 and γ_2 represent the normal angles for the intersection points respectively. With the distance d of the viewpoint from the center of the sphere, the radius r of the sphere, and the ray angle ω, we have a situation in which the ambiguous cases of the law of sines can be used to find either ϵ_1 or ϵ_2. Either of these angles can then be used to find γ_1 or γ_2.

The ambiguous case of the law of sines states that ϵ_1 is obtuse, ϵ_2 is acute. Applying the law of sines obtains the acute angle ϵ_2, implying that $\epsilon_1 = \epsilon_2 - \pi$. Using the law of sines

$$\epsilon_2 = \text{asin}(\frac{d}{r} \sin \omega). \tag{3}$$

Using the fact ω, ϵ_2, and γ_2 are three angles of a triangle, $\gamma_2 = \pi - \omega - \epsilon_2$. Solving for ϵ_2, we get

$$\gamma_2 = -\text{asin}(\frac{d}{r} \sin \omega) - \omega + \pi, \tag{4}$$

which signifies the second intersection point. We also know that $\gamma_1 = \pi - \omega - \epsilon_1$ since these three angles are of the same triangle as well. Again using the ambiguity of the law of sines, we substitute $\epsilon_1 = \pi - \epsilon_2$ into the above equation to obtain $\gamma_1 = \epsilon_2 - \omega$. Substituting Equation 3 into this new equation obtains the first intersection point:

$$\gamma_1 = \text{asin}(\frac{d}{r} \sin \omega) - \omega, \tag{5}$$

When Equation 5 is implemented on the GPU, the ratio $c = d/r$ can be precomputed. Additionally, let $u = \widehat{r} \cdot \widehat{k}$. Then substituting Equation 2 for ω in Equation 5 results in

$$\gamma_1 = \text{asin}(c\,u) - \text{asin}(u). \tag{6}$$

Sphere Normal. Next we use Equation 6 and the ray angle to find the normal angle at the front facing intersection point for both reference spheres.

The minimum value for a ray angle is zero, corresponding to the nadir, and the maximum value for a ray angle is equal to the tangent angle, which can be found using Equation 9. Let $b = \zeta \tau_0$ be the beginning angle of the range, where the result of interpolation is entirely from the primary reference sphere. ζ is defined as a user-defined threshold for blending from $[0, 1]$. The blending factor is calculated with the following equation:

$$f = (\omega - \tau_0)/(b - \tau_0) \tag{7}$$

where ω is the ray angle.

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.

This blending factor can be used to intersect the secondary reference sphere by mixing between the beginning ray angle and the angle of the ray tangent to the secondary reference sphere. 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 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 $\widehat{k}$ and $-\widehat{n}$ 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:

$$\widehat{s} = -\widehat{n} \cos \gamma + \widehat{k} \sin \gamma \tag{8}$$

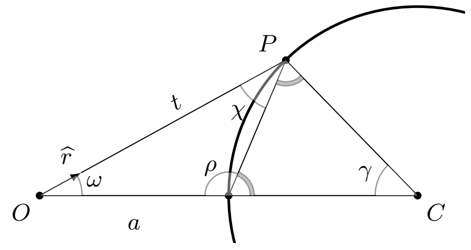

Fig. 4. The intersection position in eye space.

Position. To maintain precision, the position is generated in sampling camera space as opposed to world space. Our method of generating positions in this space uses the law of sines. Consider Figure 4. a, $\widehat{r}$, ω, and γ are known. Although there are two reference spheres, 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 γ describes a position on the planet independent of either reference sphere. The angle χ is calculated from γ using

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

When we observe that any triangle formed from an arc is an isosceles triangle, the angle ρ is then

$$\rho = \frac{\pi}{2} + \frac{\gamma}{2}$$

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

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

t and $\widehat{r}$ are then used to calculate the position in sampling camera space using $P = t\widehat{r}$.

4 Reference Spheres

We select two reference spheres, the primary and secondary reference sphere, each defined by a center and a radius. The primary reference sphere minimizes stretching and avoids shrinking while the secondary reference sphere produces intersection points that represent the occluded area. The PGM step projects the grid points onto both spheres and interpolates the results. Because the center of the planet is the same for both spheres, we are concerned only with the radii.

Sampling Issues. In preparation for PGM, the reference spheres ensure that no stretching or shrinking occurs and that the pertinent area is visible. The sampling camera is tasked with ensuring that all of the pertinent rays are generated.

Stretching occurs when grid points are excessively displaced toward the viewer. The greater the displacement distance, the larger the triangles appear in screen space.

The effects of stretching are apparent in the extreme case where the mesh is maximally displaced. This can result in parts of the mesh extending beyond the viewing frustum. Since our objective is to generate approximately pixel-sized triangles, the displacement distance must be minimized by choosing an appropriate reference sphere.

Shrinking occurs when grid points are displaced away from the viewer. Therefore, edges of the mesh can fall within the view frustum. Although shrinking can be solved by using the minimum sphere as the reference sphere, this results in a large amount of stretching.

The *pertinent area* is the geographic area of the planet that affects the final image. We must take care not to undersample or oversample this area. Rays that intersect the pertinent area on the sphere are known as *pertinent rays.* Using the appropriate reference sphere and sampling the entire pertinent area allows mountains within the occluded area to be seen in the visualization.

Primary Reference Sphere. To select the radius of the primary reference sphere, we can use the minimum visible planetary radius from the previous frame. During the rasterization step, 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. After the min mipmap in constructed, the texel in the highest level of the mipmap is read from the GPU to the CPU and saved for the next frame. Obtaining the radius this way results in a more robust rendering that minimizes stretching while avoiding shrinking.

Secondary Reference Sphere. To calculate the radius, 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 5. Let the angle τ be the angle of the tangent ray $\widehat{t}$ to the primary reference sphere $\mathbf{P}$. The ray intersects the maximum sphere $\mathbf{M}$, and the vector $\widehat{s}$ represents the sphere normal at 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.

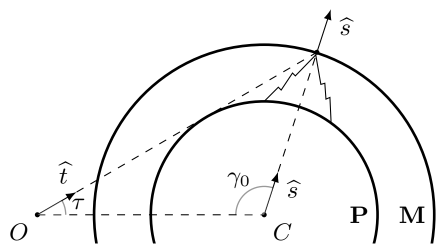

Fig. 5. The upper bound of the normal angle.

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 6. 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

$$\tau = \operatorname{asin}(r/d). \tag{9}$$

To calculate the upper bound of the normal angle γ_0, we use the tangent angle τ and Equation 4 to calculate the normal angle at the back facing intersection of the tangent ray with the maximum sphere.

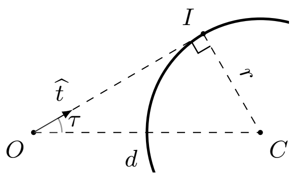

Fig. 6. The tangent angle.

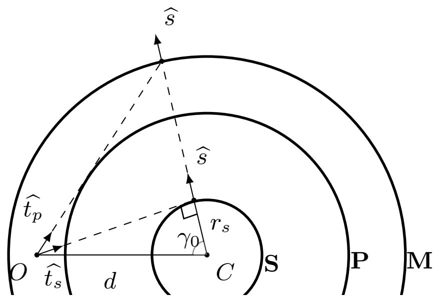

Fig. 7. The radius of the secondary reference sphere.

As Figure 7 shows, the ray $\widehat{t_p}$ is tangent to the primary reference sphere **P** and intersects the maximum sphere **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 **S** should be the sphere with a tangent 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

$$r_s = d\,\cos\,\gamma_0. \tag{10}$$

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 8 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.

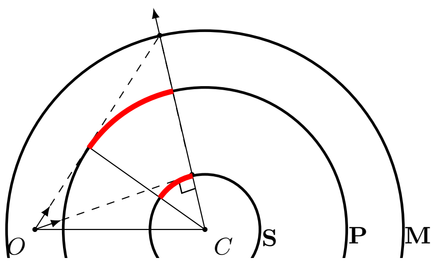

Fig. 8. The occluded area is visible on the secondary reference sphere.

The secondary reference no longer exists when Equation 10 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.

5 Sampling Camera

Since the purpose of the sampling camera is to capture only the geographic area that could affect the final image, the sampling camera is crucial to the efficiency and visual fidelity of our approach. Similar to the viewing camera, the sampling camera is defined by a view matrix and a projection matrix. These two matrices are required for projecting the grid and rasterizing the resulting mesh.

Pertinent Rays. To ensure that the sampling camera minimizes rays that miss the sphere and maximizes pertinent rays, we intersect the set of eye rays defined by the viewing camera and the entire set of rays that intersect the sphere from the viewer's current position.

The former is determined by the viewing frustum. The latter is determined by an infinite cone emanating from the viewpoint and containing the sphere entirely. To simplify intersecting these two regions, we approximate the planet cone with a frustum, called the planet frustum, which contains the planet. Every side of this frustum is tangent to the sphere.

A third area that needs to be included is the region below the viewing frustum between the nadir and where the frustum intersects the planet, as shown in Figure 9.

To generate the sampling camera, we intersect the viewing frustum with the planet frustum and include the nadir.

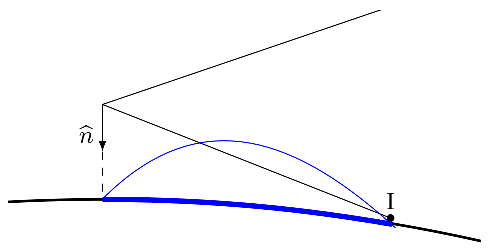

Fig. 9. The area below the viewing frustum could include terrain that should intersect the frustum.

Frustum-Frustum Intersection. The algorithm for intersecting frusta is simplified by the observation that the viewing frustum and the planet frustum share an apex, the viewpoint. The algorithm represents each frustum with respect to the corners where its sides meet, and defines each corner using a ray. The viewing frustum's corner rays can be obtained from the attributes used to define the frustum, *i.e.*the left, right, bottom, and top values which are in eye space. The planet frustum corner rays are defined by generating a frustum that is oriented so that its up direction is aligned to the plane of the viewing camera's up and forward directions.

Let frustum **A** be the viewing frustum, **B** be the planet frustum, and **C** be the intersection of **A** and **B**. Given the corner rays of **A** and **B**, the corner rays of **C** can be determined by applying two rules. First, any corner ray of **A** that is in **B** or of **B** that is in **A** is a corner ray of **C**. Second, possible corner rays of **C** result from the intersection of the planes of **A** with the planes of **B**. Any such ray that is bounded by both **A** and **B** is a corner ray of **C**.

The planes of a frustum are the planes that contain each side of the frustum. These planes are defined by their normals, which are obtained from the cross product of the corner rays.

Using these two rules, our algorithm is as follows:

1. Find the rays of **A** that exist within **B**.
2. Find the rays of **B** that exist within **A**.'
3. Calculate the normals of the planes of **A** and **B**.
4. Intersect the planes of A with the planes of B.
5. For each resulting ray, if the ray is bounded by **A** and **B** it is a corner ray of **C**.

This algorithm requires the ability to determine whether a ray is bounded by a frustum and the ability to intersect planes. To determine whether a ray is bounded by a frustum, we test the ray's angle to each plane of the frustum. Given the normal $\widehat{n}$ to a plane and a ray direction $\widehat{r}$, the ray is on the positive side of the plane if $\widehat{n} \cdot \widehat{r} > 0$. When generating plane normals for the frusta, we ensure that each plane normal points inward to the frustum. The second requirement is to intersect to the planes. The ray that lies in the intersection of two planes can be determined by the cross product of the planes normals.

In the case that there is no intersection between the frusta, as when the viewer looks away from the planet, we use the corner rays of the planet frustum.

We carry out the intersection in a special coordinate system based on the viewing camera. This coordinate system is a rotated version of the viewing camera such that the up direction lies in the plane of the view direction and the nadir. The purpose of this is to generate a sampling camera that is oriented such that the top edge of the sampling frustum is aligned with the horizon. This minimizes the number of rays that miss the sphere, especially when the viewer is close to the surface of the planet.

View Matrix. The output of frustum-frustum intersection is a set of rays that describes the corners of the intersection frustum.

To build the view matrix of the sampling camera, we first need to choose a view direction based on the intersection rays. The view direction should take into account all of the intersection rays, since a skewed view direction leads to nonuniform distribution of grid points when the projective grid is stretched across the viewport of the sampling camera. To obtain the view direction, we first set its x-component to zero, since the previously described rotated space already aligns the top of the camera to the horizon. We then take the average of the y- and z-components of all intersection rays. We include the nadir in this average in order to include the area below the viewing camera on the planet, as described earlier.

The view direction and the nadir are then used in a series of cross products to build a basis which defines the coordinate system of the sampling camera. The three vectors of this basis are used to create the sampling camera's view matrix.

Projection Matrix. The frustum of the sampling camera can be described using the standard left, right, top, bottom, near, and far values used to create a projection matrix. These values relate to the corners of the frustum on the user-chosen near plane.

We obtain these values by intersecting all of the intersection rays, along with the nadir, with an arbitrary near plane. This is carried out in coordinates relative to the sampling camera, which were defined in the previous section. The user-chosen near plane is perpendicular to the view direction within that coordinate system. We then find the minimum and maximum $x-$ and $y-$components of the resulting intersections on the near plane. The output is the left, right, bottom and top values. The distance to the far plane, like the near plane, is chosen arbitrarily. Thus, the near and far values can be the same values used to generate the viewing frustum's projection matrix.

6 Deferred Texturing

For height map composition, we must iterate over all height maps and combine the height values for each grid point. The accumulated heights are stored in a buffer of 32-bit floating points that has the same dimensions as the projective grid. This process can be performed on the GPU by using the map equations from Section 7. Color and surface normal data is composited in a similar manner, expanded for three channels of data.

The sphere normals, positions, and accumulated heights buffers describe the final terrain mesh. 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 [15]. This accomplishes performance throttling by allowing the grid size to differ from the screen size.

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. Each vertex then has an (x, y) position equal to the corresponding grid point's texel coordinates within the three buffers. By sampling each buffer, we can determine the vertex's final position, which is displaced along the sphere normal.

To mitigate the issues of 24-bit depth buffer precision, we perform a two-pass rasterization approach [16], rendered with a near frustum and a far frustum. If secondary objects are to be rendered, a stencil buffer can be maintained to track which pixels are occluded by the near terrain.

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 and inner atmospheric effects are given by Sean O'Neil [17].

7 Map Projections

For the composition of different datasets on the GPU, we must transform the normals of the sphere into texture coordinates. Using the buffer of normals, we can calculate the geographic location of each fragment by using a simple mapping from Cartesian to spherical coordinates.

Once the geographic coordinates have been determined, they must be transformed into projection coordinates using either an equirectangular or polar stereographic projection. We modify the forward mapping equation for equirectangular projection [18] by removing the radius. This serves to reduce floating point errors from a large planetary radius.

$$\begin{aligned} p &= (\lambda - \lambda_0) \cos \phi_0, \\ q &= \phi, \end{aligned}$$

where λ and ϕ are the longitude and latitude components of the geographic coordinates and λ_0 and ϕ_0 are the center longitude and latitude of the projection.

For polar stereographic projection, a slightly different equation must be used.

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

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, calculated using

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

We have modified the common equation [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. Also, note that again the radius has been removed from this equation for the same reason as the equirectangular projection equation.

Once the projection coordinates for a fragment have been obtained, the texture coordinates can be calculated. Using GDAL [19], we can determine the projection coordinates of the corners of a given dataset on the CPU. With this information, the width and height of the dataset can be calculated in projection coordinates. Assuming a projection point P (defined by the above projection equations) an upper-left dataset coordinate U, and the dimensions of the dataset D, we can determine the (s,t) texture coordinates by

$$(s,t) = \frac{P - U}{D}$$

giving us texture coordinates in the range of $[0,1]$.

8 Results

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. For these tests, the amount of texture data in GPU memory is kept constant.

Table 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. Dataset sizes are reported in megabytes (MB).

In order to understand the performance of separate parts of the algorithm, we measure the runtime of the PGM step as well as the entire algorithm. In order to see how the view affects the efficiency of the algorithm, we use three different views shown in Figure 10. We test a global view of the planet, a view close the planet using low-resolution data, and finally a view close to the planet using high-resolution data of Victoria Crater.

Finally, the problem size is varied with respect to two variables: the grid size and the screen size.

Name	Scale	Width	Height	Size
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 1. Information about the datasets used for timing.

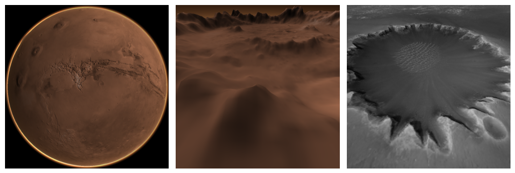

Fig. 10. Three tested views

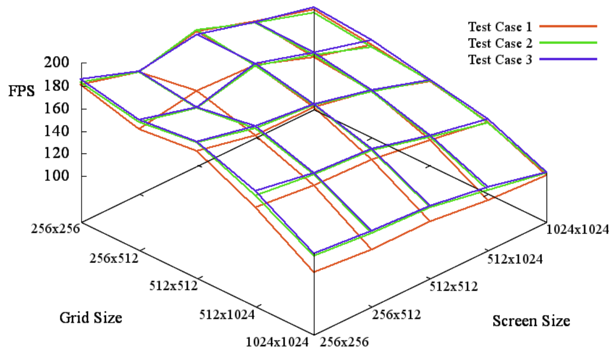

Fig. 11. Total FPS for all three cases.

Performance Analysis. Figure 11 graphs the framerate in frames per second (FPS) for all three cases. As is evident from the graph, all three cases respond similarly to changes in grid 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 both limited to the same textures and cover a smaller area. This leads to more of a performance benefit from the caching of texture reads.

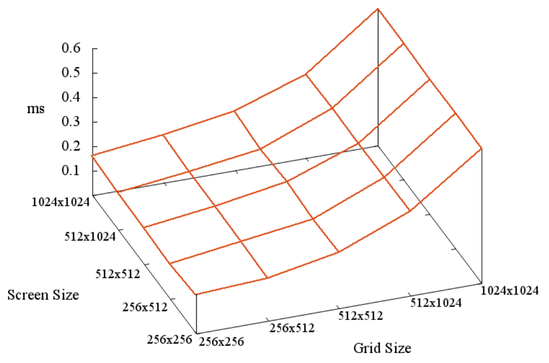

Fig. 12. The execution time in ms of the PGM step for the first test case.

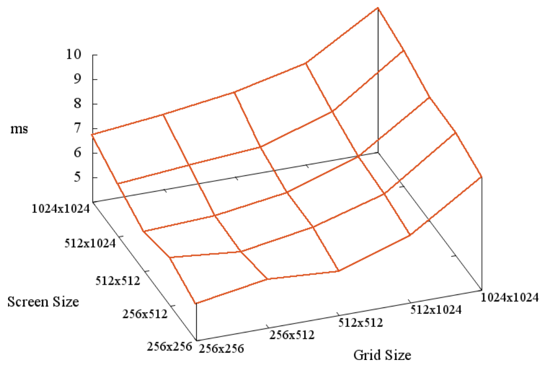

Fig. 13. The execution time in ms of the entire algorithm for the first test case.

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 12, 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 13. 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.

9 Conclusions and Future Work

Using a spherical approach to PGM, we have presented a planetary terrain renderer that accurately handles differently projected data. Additionally, we have shown how reducing the ray-sphere intersection to a two-dimensional cross section can greatly optimize the ray casting process.

To improve our algorithm, we suggest sorting the datasets into a spatial hierarchy therefore allowing for visibility tests to be performed by the CPU, possibly eliminating entire composition passes. Additionally, finding the radius of the primary reference sphere through min mipmapping is effective but slow due to the reduction performed on the GPU. To improve this, we suggest the use of CUDA [20] which allow for easier management of the shader cores. Lastly, we would like to provide a method for streaming data from the disk to RAM to GPU memory so that high-resolution data can be rendered.

10 Acknowledgments

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

References

1. Kooima, R., Leigh, J., Johnson, A., Roberts, D., SubbaRao, M., DeFanti, T.: Planetary-scale terrain composition. IEEE Transactions on Visualization and Computer Graphics **15** (2009) 719–733
2. Duchaineau, M., Wolinsky, M., Sigeti, D.E., Miller, M.C., Aldrich, C., Mineev-Weinstein, M.B.: ROAMing terrain: Real-time optimally adapting meshes. In: VIS '97: Proceedings of the 8th conference on Visualization '97, IEEE Computer Society Press (1997) 81–88
3. Losasso, F., Hoppe, H.: Geometry clipmaps: terrain rendering using nested regular grids. In: SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, ACM (2004) 769–776
4. Asirvatham, A., Hoppe, H.: Terrain rendering using GPU-based geometry clipmaps. In: GPU Gems 2. Addison-Wesley Professional (2005) 27–46
5. Dick, C., Krüger, J., Westermann, R.: GPU ray-casting for scalable terrain rendering. In: Proceedings of Eurographics 2009–Areas Papers, Eurographics Association (2009) 43–50
6. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., Scopigno, R.: Planet-sized batched dynamic adaptive meshes (P-BDAM). In: Proceedings of the 14th IEEE Visualization 2003, IEEE Computer Society (2003) 147–154
7. Clasen, M., Hege, H.: Terrain rendering using spherical clipmaps. In: EuroVis06: Joint Eurographics - IEEE VGTC Symposium on Visualization, Eurographics Association (2006) 91–98
8. Hinsinger, D., Neyret, F., Cani, M.: Interactive animation of ocean waves. In: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation, ACM (2002) 161–166
9. Johanson, C.: Real-time water rendering. Master's thesis, Department of Computer Science, Lund University (2004)
10. Bruneton, E., Neyret, F., Holzschuch, N.: Real-time realistic ocean lighting using seamless transitions from geometry to BRDF. In: Computer Graphics Forum, Wiley Online Library (2010) 487–496

11. Livny, Y., Sokolovsky, N., Grinshpoun, T., El-Sana, J.: A GPU persistent grid mapping for terrain rendering. The Visual Computer **24** (2008) 139–153
12. Löffler, F., Rybacki, S., Schumann, H.: 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, University of West Bohemia (2009) 47–54
13. Schneider, J., Boldte, T., Westermann, R.: Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In: Vision, modeling, and visualization 2006: proceedings, November 22-24, 2006, Aachen, Germany, IOS Press (2006) 145–153
14. Fourquet, E., Cowan, W., Mann, S.: Geometric displacement on plane and sphere. In: Proceedings of graphics interface 2008, Canadian Information Processing Society (2008) 193–202
15. Kooima, R.: Planetary-scale Terrain Composition. PhD dissertation, Department of Computer Science, University of Illinois at Chicago (2008)
16. Brandstetter, W.E.: Multi-resolution deformation in out-of-core terrain rendering. Master's thesis, Department of Computer Science and Engineering, University of Nevada, Reno (2007)
17. O'Neil, S.: Accurate atmospheric scattering. In: GPU Gems 2. Addison-Wesley Professional (2005) 253–268
18. Eliason, E.: Hirise catalog. http://hirise.lpl.arizona.edu/PDS/CATALOG/DSMAP.CAT (Accessed July 21, 2010) (2007)
19. GDAL. (http://www.gdal.org (Accessed July 21, 2010))
20. Nvidia: CUDA. http://www.nvidia.com/object/cuda_home_new.html (Accessed September 13, 2010) (2007)