An Implementation of Terrain Geomorphing in the Vertex Shader for Synthetic Planetary Surfaces

by

Lawrence W. Colagiovanni

S.B. C.S., M.I.T, 2004

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
NOV 13 2008
LIBRARIES

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

February 2008

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
~~OCT 23 2008~~
LIBRARIES

©2008 Massachusetts Institute of Technology
All rights reserved.

Author_____
Department of Electrical Engineering and Computer Science
February 1, 2008

Certified by_____
Bryan Wasileski
Draper Laboratory Supervisor

Certified by_____
Jovan Popović, Associate Professor
M.I.T. Thesis Supervisor

Accepted by_____
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

ARCHIVES

This page intentionally left blank.

An Implementation of Terrain Geomorphing in the Vertex Shader for Synthetic Planetary Surfaces

by
Lawrence W. Colagiovanni

Submitted to the
Department of Electrical Engineering and Computer Science

February 1, 2008

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The purpose of this research is to develop the three-dimensional models of planetary surfaces which can be used in the test environments for the Vision-Based Navigation Systems' (VBNS) terrain recognition and navigation algorithms. The VBNS will fly a trajectory over the planet, taking pictures along the flight and dynamically updating the imagery passed to the VBNS. The pictures will be generated based on the current camera angle and attitude. Testing in this more realistic, arbitrary planetary environment will help ensure the most robust algorithms for the VBNS. A terrain generation tool is implemented that allows users to create synthetic planetary terrains. Users are able to arbitrarily place terrain features (e.g. craters, mountains, and boulders), textures, and lighting to create realistic planetary surfaces. Creating these complex, three-dimensional models presents its own problem: a significantly larger amount of data has to be processed. To solve this problem, geomorphing is implemented and incorporated into the simulations. Geomorphing refers to the process of seamlessly switching between different levels-of-detail of data, which are data sets with different resolutions; the goal is to display the appropriate level-of-detail of data based on where the VBNS is currently located. Simply switching between the different data sets causes discontinuities and visual anomalies, which affect the algorithms of the VBNS. Therefore, the switching has to be done seamlessly, so that the VBNS cannot tell there was a change in the data sets being rendered. The test results indicate that geomorphing helped to reduce the discontinuities in the simulations that occur when simply switching between the different levels-of-detail.

Technical Supervisor: Bryan Wasileski
Title: Senior Member of the Technical Staff, Draper Laboratory

Thesis Supervisor: Jovan Popović
Title: Associate Professor, MIT Department of Electrical Engineering and Computer Science

This page intentionally left blank.

Acknowledgement

February 1, 2008

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under the Planetary Landing Internal Research and Development Project (Phase II, 2116-002) and the LSAM Cockpit Internal Research and Development Project (21790-002).

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

(Author's Signature)

This page intentionally left blank.

Table of Contents

ABSTRACT **3**

ACKNOWLEDGEMENT **5**

TABLE OF CONTENTS **7**

LIST OF FIGURES **9**

INTRODUCTION **11**

1.1 PROBLEM STATEMENT 11
1.2 GOALS 13
1.3 THESIS OUTLINE 13

2 PREVIOUS RESEARCH **15**

2.1 TERRAIN GENERATION 15
2.2 RENDERING TERRAIN DATA 16
2.2.1 Culling *16*
2.2.2 Geomorphing *17*

3 DESIGN OVERVIEW **21**

3.1 TERRAIN GENERATION TOOL 21
3.2 GEOMORPHING SIMULATION 22

4 TERRAIN GENERATION TOOL **25**

4.1 TERRAIN ATTRIBUTES 25
4.2 TERRAIN MESH 25
4.3 TERRAIN FEATURES 27
4.3.1 Crater *27*
4.3.2 Control Points Features *29*
4.3.3 Boulders *31*
4.3.4 Distribution Curves *31*
4.3.5 Randomness *32*
4.3.6 Adding and Removing Features *33*
4.4 TEXTURES 33
4.5 LIGHTING 34
4.5.1 Overview *34*
4.5.2 Light Sources *34*
4.5.3 Light Maps *35*
4.6 TERRAIN DATABASE GENERATION 35

5 GEOMORPHING TOOL **39**

5.1 PRE-PROCESSING 39
5.1.1 Geometry Creation *39*
5.1.2 Shaders *41*
5.2 SIMULATION 43
5.2.1 Supported Simulation Types *43*
5.2.2 Geomorphing Logic *43*

5.2.3 Capturing Images *44*

6 TESTING **45**

6.1 METHODOLOGY 45

6.2 RESULTS 46

7 FUTURE RESEARCH **49**

8 CONCLUSION **51**

ACKNOWLEDGEMENTS **53**

REFERENCES **55**

List of Figures

Figure 1: Steps in vision-based navigation simulation 11
Figure 2: Image of the Moon from 100 meters away (left) and 10 meters away (right) 12
Figure 3: Images of the Moon with gaps in the data (left) and inconsistent lighting conditions (right) 12
Figure 4: Sample terrain generated by 3DEM 15
Figure 5: Patch of terrain generated using fractals 15
Figure 6: Sample terrain generated by Crater Maker 16
Figure 7: Approximating meshes using irregular grids 18
Figure 8: Two levels-of-detail and the vertices (in red) removed 18
Figure 9: T-vertices (displayed in red) for levels-of-detail 19
Figure 10: Adding triangles to remove t-vertices 19
Figure 11: Vertex (red) morphing to position in coarser level (dotted circle) 20
Figure 12: Terrain generation tool overview 22
Figure 13: Application and shaders 23
Figure 14: Terrain mesh attributes 26
Figure 15: Mesh with four tiles and n equal to one 26
Figure 16: Tiles with different levels-of-detail 27
Figure 17: Mesh with uneven number of vertices 27
Figure 18: Crater attributes 28
Figure 19: Impact elevation and direction angles 28
Figure 20: Crater with exponential decay(left) and linear decay (right) 29
Figure 21: Examples of a mountain, valley, and plateau and their respective control points 29
Figure 22: Example of interpolation when constructing curves 30
Figure 23: Control points used to construct four mountains 31
Figure 24: Example of boulder distribution 32
Figure 25: Control points feature with and without roughness 33
Figure 26: Sample texture 34
Figure 27: Surface and corresponding light map 35
Figure 28: Three levels-of-detail 36
Figure 29: 1st, 4th, and 6th levels-of-detail with resolution 2.3 km, 18.4 km, and 73.6 km, respectively 36
Figure 30: Triangles for the 4th and 6th levels-of-detail 36
Figure 31: Active and morphing ranges 37
Figure 32: Corner faces (blue), outer faces (yellow), and inner faces (white) 39
Figure 33: Triangular strips (each color indicates a different strip) 39
Figure 34: Inserting geometry to eliminate gaps 40
Figure 35: Possibilities for vertices geomorphing 40
Figure 36: Vertex (red) morphing to position in coarser level (dotted circle) 41
Figure 37: Adjusting levels-of-detail 43
Figure 38: Test surface 45
Figure 39: Results 47

This page intentionally left blank.

Introduction

1.1 Problem Statement

This research supports Draper Laboratory's Planetary Landing Independent Research and Development Project (IR&D), which focuses on developing planetary surface recognition and navigation algorithms for vision-based navigation systems (VBNS).

VBNS work by first using onboard cameras to periodically capture images of a planetary surface while the system is in flight. The surface images are then analyzed and compared against a database of known geographic features to determine the vehicle's position, velocity, and attitude. As the vehicle flies along its trajectory, the flight computer and motion sensors track the vehicle's position, continually updating the inertial navigation system (INS). The guidance system evaluates the vehicle's new state information, modifying the control system as needed, ensuring that the vehicle remains on the desired flight path. VBNS are primarily used because they are able to provide independent navigational information when external systems, such as GPS and ground-based radar, are unavailable.

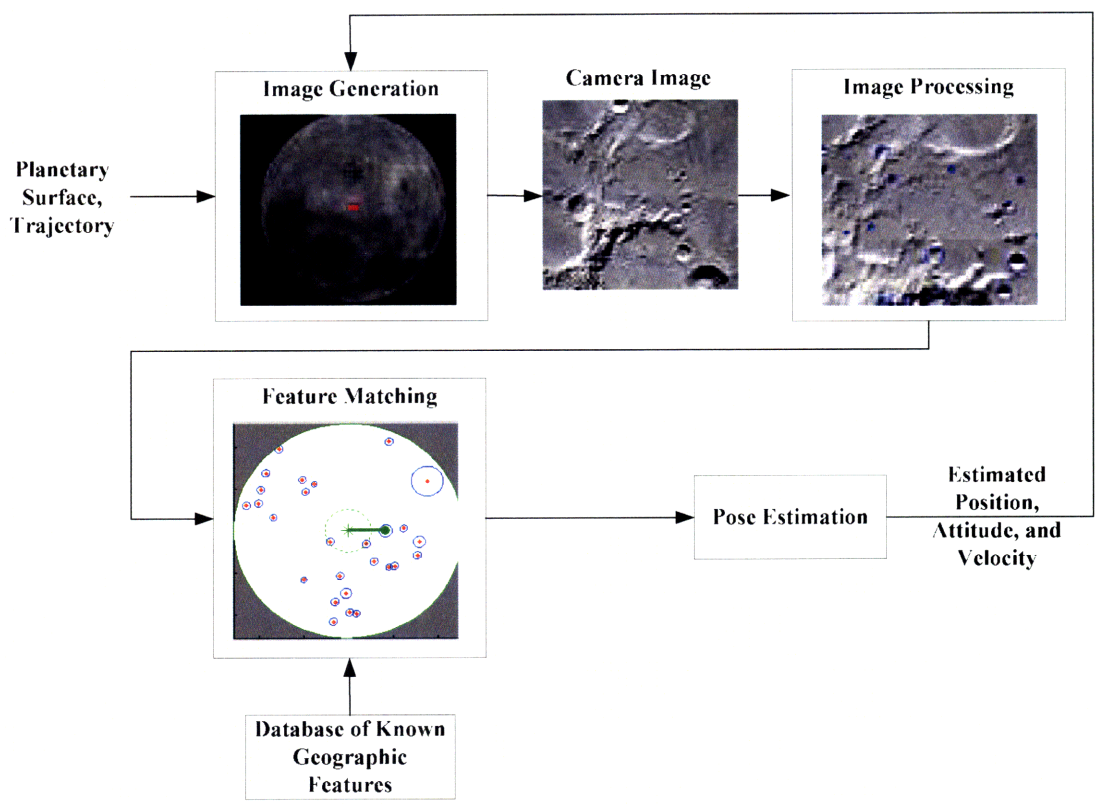

Figure 1: Steps in vision-based navigation simulation

To test VBNS, the recognition and navigation algorithms are run on static imagery from a planetary surface; the static image to display is chosen based on the location of the VBNS. Static images are used because adequate three-dimensional models of the planetary surfaces do not exist. Using the static imagery presents two problems: the algorithms are developed against a fixed data set and the imagery is generally of poor quality.

There is currently a limited amount of imagery of planetary surfaces available, with most images coming from the Moon and Mars. The surfaces of the Moon and Mars share similar topographical characteristics, with craters primarily covering both. Additionally, the camera's position, orientation, and field-of-view are also fixed in the images, limiting the perspective of the VBNS. Therefore, when using these images, the VBNS are tested and tuned based to a limited set. Having a wider variety of planetary surfaces available for testing would greatly improve the tuning of the VBNS.

One issue with the image quality is that the resolution is too coarse. As a VBNS approaches the surface in a test simulation, the image becomes more blurred, and the VBNS is no longer able to detect features on the surface, as seen in figure 2. This is especially problematic for VBNS that are used for detecting and avoiding small hazards on the planetary surface, such as boulders.

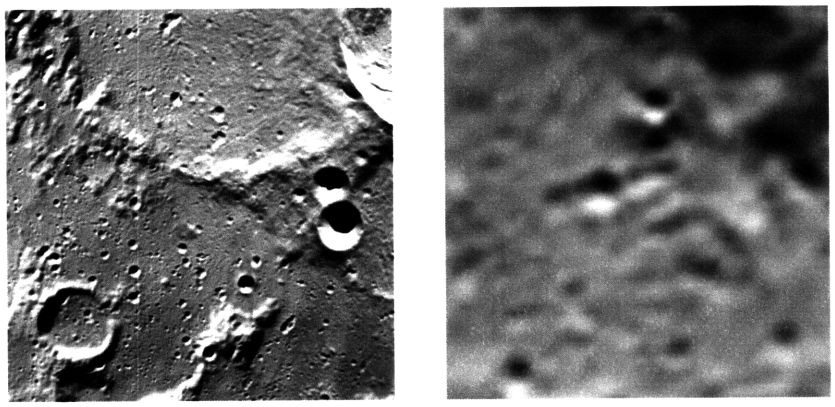

Figure 2: Image of the Moon from 100 meters away (left) and 10 meters away (right)

The planetary surface data is gathered using satellites. As the satellites orbit the planets, they either do not measure parts of the surface or some of the data is lost in transmission, leading to no information about parts of the surface, as shown in figure 3. Additionally, lighting conditions differ depending on when the satellite orbits the planet. Consequently, when comparing the images of two locations next to each other on the surface, one image may be much lighter than the other, as shown in figure 3.

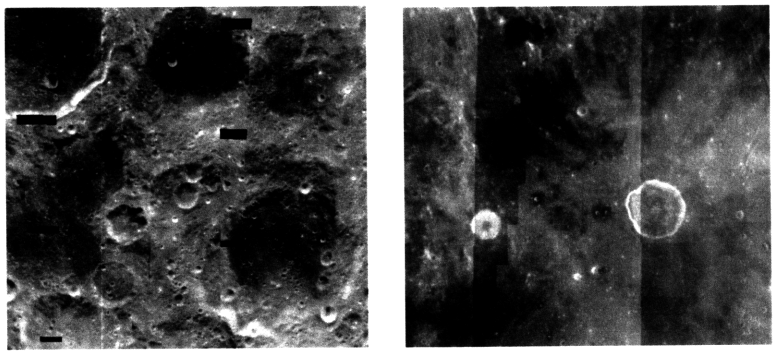

Figure 3: Images of the Moon with gaps in the data (left) and inconsistent lighting conditions (right)

1.2 Goals

The purpose of this research is to develop the three-dimensional models of planetary surfaces which can be used in the test environment for the VBNS algorithms. Rather than using static images, the VBNS will fly a trajectory over the planet, taking pictures along the flight and dynamically updating the imagery passed to the VBNS. The pictures will be generated based on the current camera angle and position, rather than being fixed. Testing in this more realistic planetary environment will help ensure the most robust algorithms for the VBNS.

A terrain generation tool is implemented that allows users to create synthetic planetary terrains. Users can place craters, mountains, valleys, boulders, etc. on the planetary surface. After creating the surface, the user can apply textures to it, making it look more realistic. The user can also either supply a light map or add a light source to simulate the effects of lighting on the surface. If the user does not want to create an entirely synthetic terrain, then the user can load existing planetary data and manipulate it by increasing its resolution, adding more features to fill in the gaps, and/or adding lighting to the surface. Since the user can create varied synthetic terrains, the VBNS can be tested in a broader set of environments.

Creating these complex, three-dimensional models presents its own problem: a significantly larger amount of data has to be processed as compared to when static images are displayed. To solve this problem, geomorphing is implemented and incorporated into the simulations. Geomorphing refers to the process of switching between different levels-of-detail of data, which are data sets with different resolutions. The goal is to display the appropriate level-of-detail of data based on where the VBNS is currently located. For example, if it is far away, then coarser data is rendered because the VBNS does not require as much detail in the surface. As the VBNS approaches the surface, finer data is rendered, so that the VBNS can identify objects like boulders and craters. Simply switching between the different data sets causes discontinuities and visual anomalies in the camera's images, affecting the algorithms of the VBNS. Therefore, the switching also has to be done seamlessly, so that the VBNS cannot tell there was a change in the data sets being rendered. The terrain generation tool is leveraged to create these levels-of-detail.

1.3 Thesis Outline

The thesis is composed of the following eight Chapters.

- *Chapter 1, Introduction*, discusses the problem statement and the contributions that this research aims to make.

- *Chapter 2, Previous Research*, explains some of the previous research completed in the areas of planetary surface generation and methods for rendering terrains with large amounts of data.

- *Chapter 3, Design Overview*, provides a high-level design summary of the final approach chosen.

- *Chapter 4, Terrain Generation Tool*, explains, in detail, the design of the terrain generation tool.

- *Chapter 5, Geomorphing Tool*, explains, in detail, the design of the geomorphing application.

- *Chapter 6, Testing*, describes the approach taken to test the geomorphing implementation, the results that were found, and the implications of those results.

- *Chapter 7, Future Research*, discusses potential extensions for future work.

- *Chapter 8, Conclusion*, offers general conclusions regarding the research and discusses how well the implementation of the terrain generation tool and geomorphing solved the challenges of the VBNS.

2 Previous Research

2.1 Terrain Generation

Currently, there are several existing tools and techniques for generating terrains. Below focuses on three types: proprietary (or commercial) applications, fractal geometry, and Crater Maker, a prototype tool built at Draper Laboratory.

Several proprietary applications exist today for creating terrains. Users can add various features, textures, and lighting. One such tool is 3DEM, which produces three-dimensional terrain scenes and flyby animations from a variety of data sources [Horne]. The terrain data and trajectory are loaded into the tool, which then produces the flyby animation. The user can then save the animation, the surface as a texture, or the terrain height map as a binary file. The tool creates very realistic surfaces, as shown in figure 4.

Figure 4: Sample terrain generated by 3DEM

3DEM and similar, existing applications are unable to solve the challenges of the VBNS though. First, they do not support loading NASA's Planetary Data System (PDS) File Format. While some tools do support this format, they are not publicly or commercially available. Therefore, the team could not correct the coarse resolution, gaps, or lighting inconsistencies with the existing data using the available tools. Second, the existing applications do not generate the levels-of-detail with varying resolutions that are needed by geomorphing.

Fractal geometry is another commonly used mechanism for generating terrains [Martz]. In this method, the user typically starts with a flat surface and specifies the number of iterations to run the algorithm. During the iterations, the height values of vertices are randomly perturbed. Some grow more than others, so that peaks, valleys, and plateaus begin to form in the terrain:

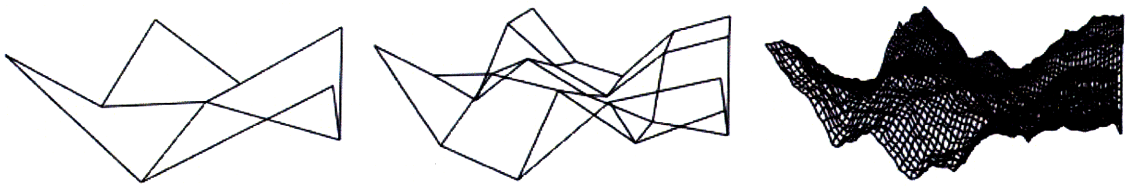

Figure 5: Patch of terrain generated using fractals

Fractal geometry creates very realistic surfaces and does so with minimal input from the user, saving the user much time. The main problem with fractal geometry, as it relates to this research,

is that it does not provide very much control for the user. For instance, if the user wanted to create a crater in a certain location, it would be difficult to construct it using fractal geometry. There are also no existing fractal geometry applications that create the levels-of-detail needed by geomorphing.

Crater Maker is a tool developed at Draper Laboratory for generating planetary surfaces with randomly distributed craters. Given a longitude range, latitude range, and crater density, the tool generates craters with varying radii and places them randomly on the surface. The craters are additive, and with a large number of craters, a realistic surface is created:

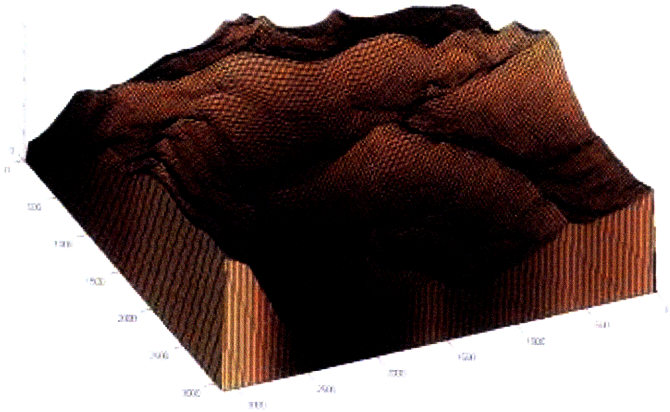

Figure 6: Sample terrain generated by Crater Maker

Crater Maker was initially designed as a prototype for generating planetary terrains, so its feature set is limited. The tool is primarily used for generating planetary surfaces with randomly placed craters. While a user could define a crater and add it to the surface, it is not what the tool is designed for. The tool also does not support adding textures or lighting to the model. Finally, it does not generate the data sets that are needed for geomorphing.

2.2 Rendering Terrain Data

Improvements in the CPU and GPU have enabled the rendering of large amounts of primitives at once. However, the data sets used in rendering planetary surfaces are large enough that even current CPUs and GPU cannot draw them at acceptable rates and quality. There are two existing techniques that are most commonly used (sometimes in combination) for reducing the amount of data that the rendering pipeline processes: culling and geomorphing.

2.2.1 Culling

Culling refers to the process of identifying faces of meshes that do not contribute to a scene and removing them from the scene. Reducing the amount of geometry processed and drawn improves the performance of the system. There are several methods used to implement culling [Hidden Surface Determination]:

- **Z-Buffering**: When objects in the scene are being drawn, the faces that compose the objects are converted to pixel values for rendering on the screen. As part of this process, a

buffer (referred to as the z-buffer) is maintained, which contains the depth of the pixel. Each object in the scene is looped through, and if its depth value is less than current value in the z-buffer, it replaces the value in the z-buffer; if not, then it is discarded.

- **Viewing Frustum Culling**: The viewing frustum refers to the area in the scene that can be seen from the current viewpoint. If an object lies outside of the viewing frustum, then it is not drawn. If any part of the object is inside of the viewing frustum, then it is drawn.

- **Back Face Culling**: When an object is constructed of faces, the faces have two sides: a front face and a back face. Since the user cannot see the back face of an object, it is not drawn.

- **Contribution Culling**: Objects in the scene that are very far away from the viewer and cannot be seen in the image are removed.

- **Ray Tracing**: Ray tracing typically applies to lighting in that rays are shot from the viewpoint to the objects in the scene to determine which are in the light and which are in shadows. In doing so, it determines which objects are behind other objects and can also be used to reduce the number of objects drawn in the scene.

- **Bounding Volumes**: Bounding volumes are used to group objects in a scene and are often used with ray tracking. When rays are cast from the given viewpoint, intersections with the bounding volumes are checked first instead of with individual polygons. If an intersection is not detected, then all of the geometry within the bounding volume does not have to be rendered. If an intersection is detected, then each polygon within the bounding volume is checked for an intersection.

Each of these culling techniques successfully removes polygons that the user cannot see or that do not contribute significantly to the scene. However, culling techniques alone are not sufficient when there is a large amount of geometry to be drawn and all of the geometry is visible from the camera's viewpoint. Given its benefits, culling is used in almost all applications today, often in combination with other techniques for reducing the amount of geometry that is processed and drawn.

2.2.2 Geomorphing

Another approach to reducing the amount of geometry that is processed and drawn is to render the appropriate amount of data based on the terrain and/or the viewpoint of the user. There are three common methods to do so: using irregular meshes, levels-of-detail, and MIP maps.

Some terrains have sections where the height values of the vertices are the same or that they follow a consistent slope. These sections of terrain can be replaced with a smaller number of triangles that are used to approximate the sections. For instance, a flat part of land or the side of a mountain that has the same slope throughout may be approximated as follows in figure 7:

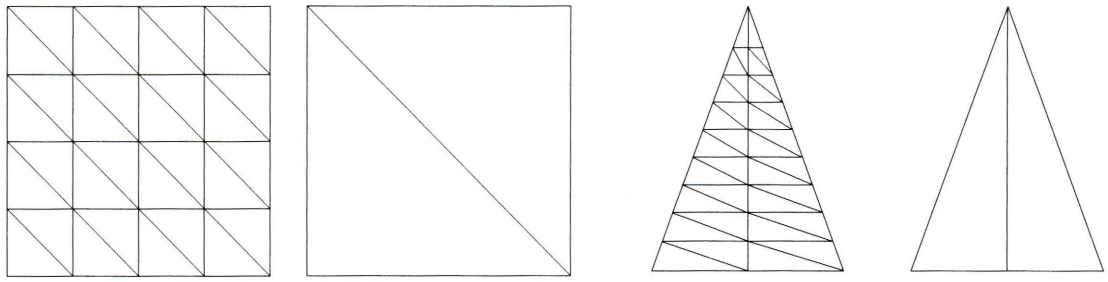

Figure 7: Approximating meshes using irregular grids

Areas of the terrain with a lot of variation continue to use a large number of triangles, so that the terrain does not lose any of its visual characteristics. The term irregular mesh is used to describe this approach since some sections of the terrain use large triangles and others use small triangles.

There are two main drawbacks to using an irregular mesh. First, in a large, irregular mesh, it can be difficult to maintain a systematic relationship between all of the vertices and triangles since they do not adhere to a regular pattern. Second, irregular meshes have little effect on varied surfaces where much of the terrain cannot be approximated using a smaller number of triangles.

Another approach is to use levels-of-detail, which are sections of the same geometry with varying resolutions [Wagner]. This approach is based on the assumption that sections of terrain that are far from the user do not need to be rendered with the amount of detail as those that are close to the user; instead, they can be approximated using a version with a coarser resolution. In the simplest approach, each level-of-detail is created by halving the resolution of the previous level-of-detail. For example, in figure 8, two levels-of-detail are pictured where 16 of the vertices have been removed from the finer level-of-detail to create the coarser one:

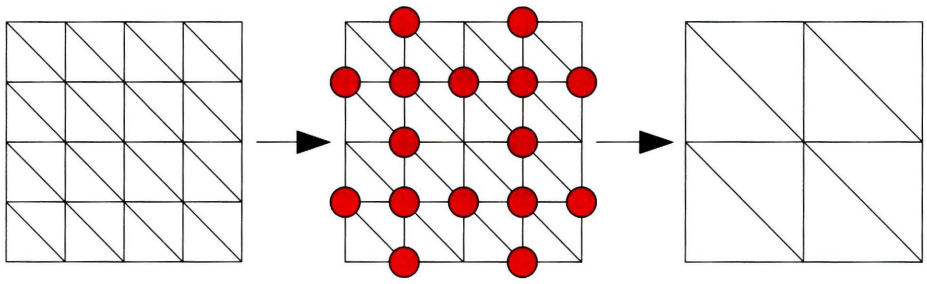

Figure 8: Two levels-of-detail and the vertices (in red) removed

MIP mapping is a similar approach to using levels-of-detail [de Boer]. MIP maps are sections of the same texture with varying resolutions. Similar to levels-of-detail, the appropriate MIP map is rendered based on the location of the camera. If the camera is far away, then a coarse version of the texture map is used. If the camera is close, then a more detailed texture is used. The principle difference between levels-of-detail and MIP maps is that levels-of-detail reduce the amount of geometry that is rendered, where as MIP maps reduce the amount of information in the textures rendered.

When using levels-of-detail or MIP maps, it possible to divide the surface into smaller levels-of-detail or MIP maps. The application can then choose which series of levels-of-detail or MIP maps to use to construct the surface based on the location of the camera. If the camera is close to

one section of the terrain and another section of the terrain is far in the distance, a finer level-of-detail can be rendered closest to the camera, while a coarser one can be rendered in the distance.

When stitching together levels-of-detail or MIP maps with different resolutions, it is usually easy to see where the two have been pieced together. When using levels-of-detail, one level-of-detail will have more vertices on the connecting edge than the other. It is probable that the height values of these extra vertices will not perfectly align with the coarser level-of-detail, causing holes in the mesh. Similarly with MIP maps, there will be more texture information in one MIP map than the other. These locations are commonly referred to as t-vertices or t-cracks and are shown in figure 9.

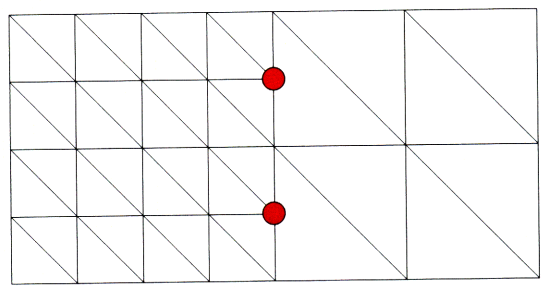

Figure 9: T-vertices (displayed in red) for levels-of-detail

Several methods have been proposed to remove t-vertices. First, some applications require that the edges be coincident, so that no discontinuities can ever exist. Another approach is to determine the minimum gap between the two tiles that can be seen when the mesh is rendered. If the gap is above this threshold, then the two levels-of-detail are not used. If the gap is below this threshold, then the two levels-of-detail are used because the user cannot see the discontinuities in the mesh. A third approach is to change the topology of the neighboring geometry, so that either the coarser level-of-detail adapts to the finer level-of-detail or vice versa by introducing new triangles to stitch the meshes together:

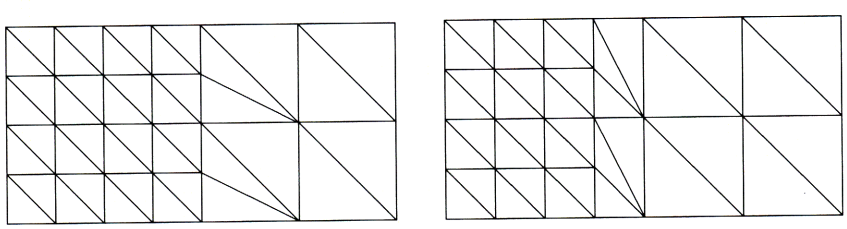

Figure 10: Adding triangles to remove t-vertices

When using levels-of-detail or MIP maps, it is possible that a level-of-detail is switched to a coarser or finer one when the camera moves further away or closer to the surface. Just switching the two levels-of-detail produces a noticeable visual effect. When the level-of-detail or MIP map is switched, either more or less geometry is suddenly rendered, depending on if it is switching to a finer or coarser level-of-detail or MIP map. This sudden change in geometry is commonly referred to as popping. To eliminate popping, geomorphing is often implemented, which refers to the process of seamlessly switching between two different levels-of-detail or MIP maps.

One geomorphing solution is to slowly change the height of the vertex, so that it approaches the height that it would have been if it was on the straight line between the two-vertices remaining in

the coarser level-of-detail [Wagner]. This process creates the seamless transition between the different levels-of-detail. To determine the amount to morph, the average height values of the two neighboring vertices are calculated. This is then subtracted from the height of the vertex to be morphed. This final value indicates the total amount that the height of the vertex will have to change when switching levels-of-detail.

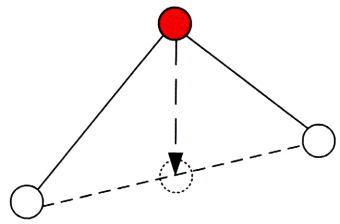

Figure 11: Vertex (red) morphing to position in coarser level (dotted circle)

Geomorphing is typically implemented at the hardware level rather than the application level because the performance would be too slow if it is done on the application level. To implement geomorphing, vertex and fragment shaders are used. Vertex shaders operate on a per-vertex basis and are responsible for changing vertex attributes, such as position, texture coordinates, and color. Fragment shaders operate on a per-pixel basis and are responsible for determining the final color of the pixels displayed on the screen. Vertex shaders are commonly used to change the height values of the vertices when using levels-of-detail, while fragment shaders are often used to morph the textures when using MIP maps.

When implementing geomorphing using levels-of-detail, the height values of the vertices change as the levels-of-detail morph [Wagner]. Consequently, the normals of the vertices also change. Since the normals are used to calculate the light values at each vertex, recalculating the light values every frame in the rendering phase would be computationally expensive. To address this issue, a single light map is often used for all levels-of-detail. Using a light map fixes the lighting conditions while the terrain morphs. The light value of the vertex is always looked up using the light map, so it becomes independent of the vertex normal. While it is not exact, it provides a reasonable approximation to the actual light values.

3 Design Overview

There are two main components of this research: the terrain generation tool and the real-time geomorphing application. The terrain generation tool is used to build three-dimensional models of planetary surfaces. When the user is done constructing a planetary surface, a terrain database is created. This database contains all of the information about the planetary surface, including the vertices of the surface mesh, levels-of-detail, textures, and lighting. The geomorphing application uses the generated databases in the real-time application.

3.1 Terrain Generation Tool

The terrain generation tool is used to build realistic, three-dimensional models of planetary surfaces. The tool is written in both Java and Matlab. The user interface is implemented in Java, while the terrain construction, calculations, etc. are implemented in Matlab.

When constructing the terrain, the user specifies the fundamental attributes of the terrain including the planet's radius, the desired longitudinal and lateral range of the output database, and the desired surface mesh resolution. The user can either create a completely synthetic terrain or start with existing planetary data. If the user chooses to create a synthetic terrain, then user starts with a clean slate (terrain) with no features, textures or lighting; the initial surface height above the planet's mean radius will be initially zero. If the user chooses to start with existing data, the tool supports loading a planetary height-map data stored in NASA's Planetary Data System (PDS) file format; a bi-linear interpolation scheme is used on the measured data to generate the surface mesh to the desired resolution.

After the initial surface is created, the user begins manipulating the surface. The tool supports adding three types of features to the terrain: craters, control point features, and boulders. To generate a crater, the user specifies basic attributes of a crater and where to place it within the terrain (e.g. location, radius, depth, wall height). Control point features involve having the user placing control points on a section of the surface with user-defined heights and weights. These points influence the heights of vertices nearby in proportion to the weight of the control point and the distance between the vertices. By strategically placing control points, the user can create features like mountains, valleys, and plateaus. To create a boulder, only the radius and location are specified. The tool also supports placing distributions of both craters and boulders. Given a longitudinal range, lateral range, and surface density, the tool randomly distributes the features over the specified range of the terrain. After each feature is created, its height-values are added to the terrain. Using this additive manner, each terrain feature becomes a separate element which can be manipulated or removed from the surface independent of the other features.

The user can also add textures to the surface as well as lighting to make the surface look more realistic. For lighting, if the user does not have an existing light map, the user can add a light source to the terrain which can be used to generate the corresponding light map.

When the user completes the surface, the terrain database is generated, which the geomorphing application will then use in its simulations. The terrain generation tool creates the coarser levels-of-detail from the finest level-of-detail. For each level-of-detail, the database contains the vertices of the surface, light map, texture map, and other information needed by the geomorphing application.

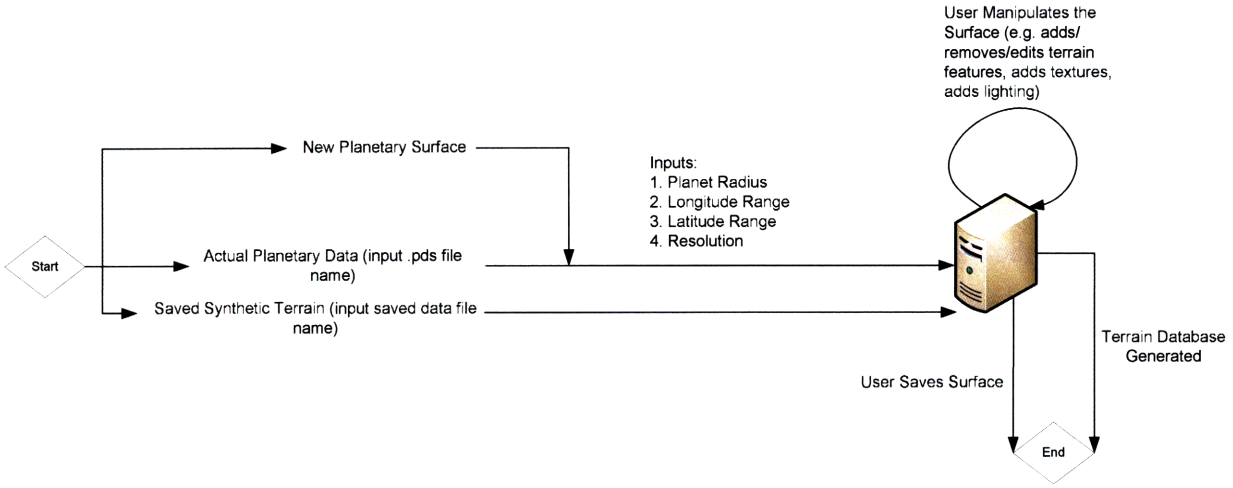

Figure 12: Terrain generation tool overview

3.2 Geomorphing Simulation

Once the terrain database is generated using the terrain generation tool, it is then used by the geomorphing application. The geomorphing application is responsible for parsing the terrain database, building the geometries of the different levels-of-detail, and then running the simulation with geomorphing. The geomorphing application is built using C++ and OpenSceneGraph (OSG). OSG is an open-source, high-performance 3D graphics toolkit, written in C++, and is used for rendering the scene. One of the benefits of using scene graph APIs, such as OSG, is that it performs the culling operations in the software by default.

The geomorphing application first parses the terrain database and builds the geometries of the different levels-of-detail. When building the geometry, the application creates the geometry that is needed when the tile's neighbors have the same, coarser, and finer tessellation levels. The tessellation level indicates which level-of-detail to use. Creating this geometry beforehand reduces the amount of computation needed during the simulation. The application also determines the values that each vertex must morph and stores this value on a per-vertex basis.

To implement the logic for geomorphing, a vertex shader and fragment shader are used. The vertex shader is responsible for morphing the height values of the vertex, while the fragment shader is responsible for rendering the textures and lighting. They are written using the OpenGL Shading Language (OGSL), which allows application programmers to create executable programs that can be loaded on the vertex and fragment processing units.

During the simulation, the application retrieves the current camera viewpoint. It then selects which level-of-detail for each tile to render. For each level-of-detail, the appropriate geometry is chosen based on the tile's tessellation level and its neighbors' tessellation levels. For instance, if one of the tile's neighbors has a finer resolution, then extra faces are added to the geometry to eliminate the t-vertices. After the geometry has been selected, the vertex and fragment shaders intercept and process the geometry.

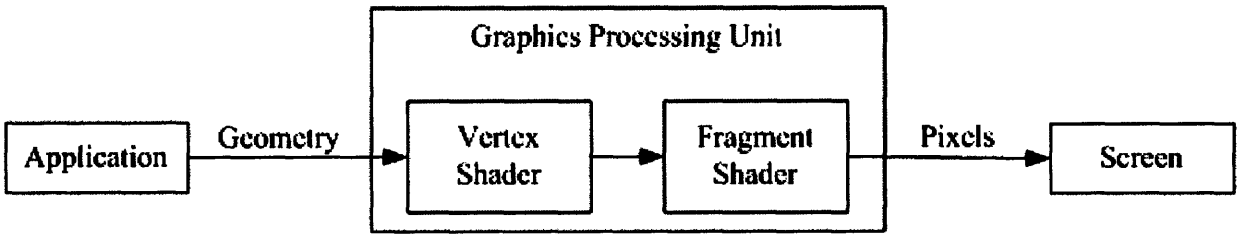

Figure 13: Application and shaders

The vertex shader determines if the vertex needs to morph. If it does, then the height value is changed the appropriate amount. The updated vertex is then passed to the fragment shader. It determines if there is a texture or light map to apply. If there are, then the fragment shader updates the color of the fragment appropriately. The scene is then rendered to the screen.

This page intentionally left blank.

4 Terrain Generation Tool

4.1 Terrain Attributes

Using the terrain generation tool, the user can either create a completely synthetic terrain or start with existing planetary data. If the user chooses to create a synthetic terrain, then the user starts with a clean slate (terrain) with no features, textures or lighting. If the user chooses to start with existing data, the tool supports loading a planetary height-map data stored in NASA's Planetary Data System (PDS) file format. The planetary data is contained in two files. The first file is the header file, which contains information about the planetary surface that the data is taken from. The following information is extracted from the file and used in the terrain generation tool:

- Minimum latitude
- Maximum latitude
- Minimum longitude
- Maximum longitude
- Number of latitude lines
- Number of longitude lines

The second file contains the height values above the planet for the longitude and latitude ranges. Using the number of latitudes and longitude, the resolution of the samples can be determined.

Regardless of type, every terrain has the following attributes:

- The **planet radius** is the radius of the planet, measured in kilometers. The planet is assumed to be spherical.
- The **longitude range** and **latitude range** are the span that the section of terrain covered. They are measured in degrees.
- The **resolution** is the distance between the vertices in the terrain and is measured in kilometers.
- The value **n** is used to determine the number of vertices per tile (explained in more detail in the Terrain Mesh section).

4.2 Terrain Mesh

After specifying the initial attributes, the terrain is generated. The longitude and latitude values are converted to kilometers, and a mesh is generated with the desired resolution. Initially, the surface height above the planet's mean radius is zero.

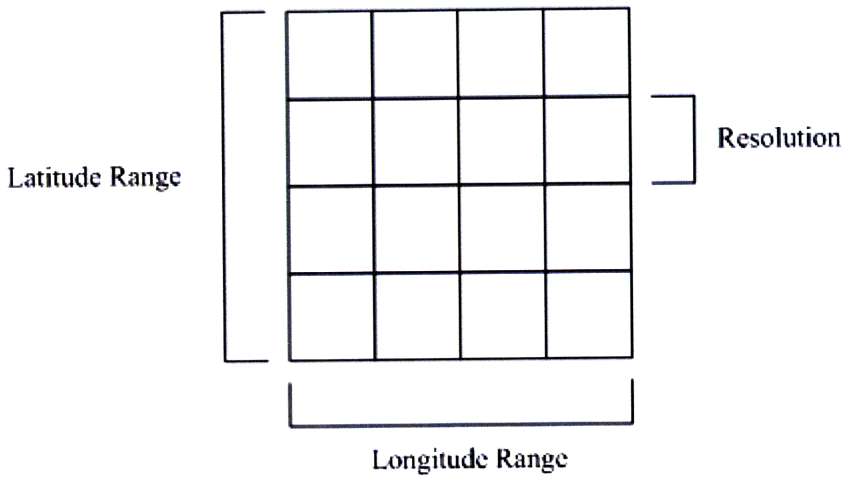

Figure 14: Terrain mesh attributes

If an actual planetary surface is chosen, a bi-linear interpolation scheme is used on the measured data to set the height values of the mesh to the values corresponding to the actual planetary data.

After creating the initial mesh, the mesh is partitioned into tiles based on the number n that the user specified: each tile contains $2^n + 1$ vertices. To enable geomorphing, the tiles must conform to this constraint. Additionally, the tiles must have the same number of vertices on each side (the number of tiles in each direction does not have to be the same though).

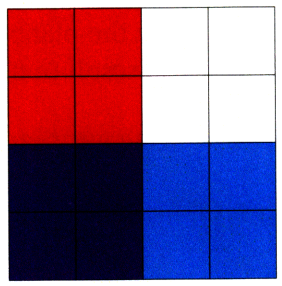

Figure 15: Mesh with four tiles and n equal to one

Splitting the mesh into tiles is useful because it enables rendering different tiles with different resolutions of data based on the viewpoint of the user. For instance, if the user was closest to the lower right in the picture below, the entire surface would have to be rendered with the finest data if there was only one tile. Having multiple tiles enables showing the finest data close to the user and coarser data elsewhere.

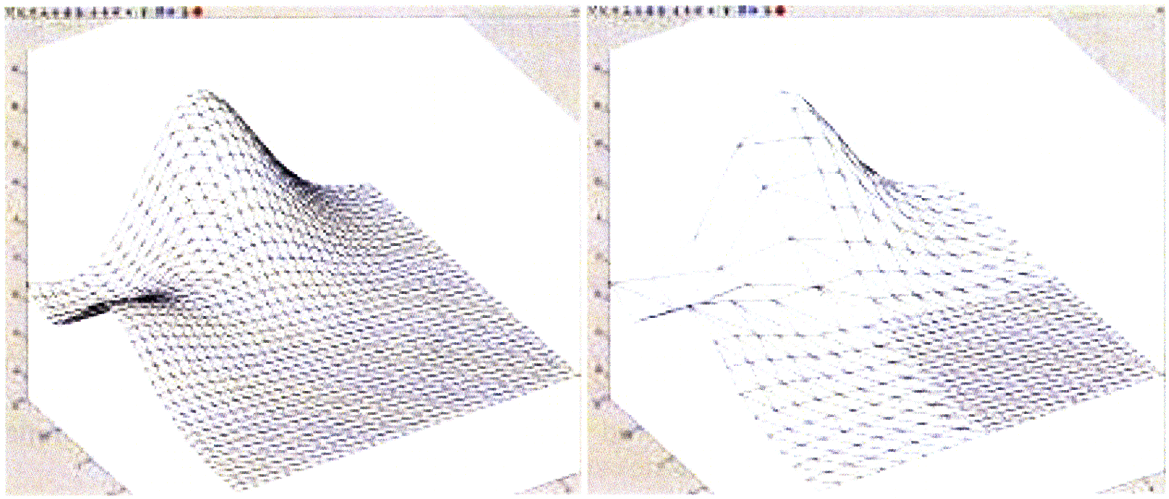

Figure 16: Tiles with different levels-of-detail

When the mesh is partitioned into tiles, it may be the case that the mesh does not contain exactly some multiple of $2^n + 1$. If this is the case, then the tiles on the end will not satisfy the constraint that each tile must have $2^n + 1$ vertices. To accommodate this problem, additional vertices are inserted into the edge tiles so that they meet the constraint.

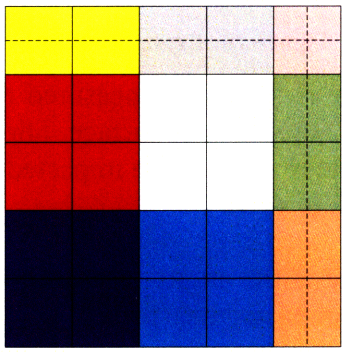

Figure 17: Mesh with uneven number of vertices

Taking this approach changes the resolution of the edge tiles, making it dissimilar to the rest of the mesh. An alternative approach would have been to alter the resolution of the mesh, so that the mesh had $2^n + 1$ vertices. This approach is not used because it can significantly change the overall resolution of the mesh.

4.3 Terrain Features

4.3.1 Crater

A crater is defined using the following attributes:

- The **center** is the longitude and latitude coordinates of the center of the crater and is measured in degrees. The center is assumed to be located on the surface of the planet, and therefore, the altitude is always set to the radius of the planet.

- The **radius** is the distance from the center of the crater to the edge of the crater wall and is measured in kilometers.

- The **wall height** is the distance from the surface of the planet to the top of the crater wall. This is measured as a percentage of the radius.

- The **crater depth** is the distance from the center of the crater to the deepest part of the crater. This is also measured as a percentage of the radius.

- The **decay type** is used to determine the shape of the curve that traveled from the top of the crater wall to the surface of the planet (away from the center of the crater). Two decay types are supported: exponential and linear. If exponential decay is used, a **decay constant** determined the shape of the exponential decay.

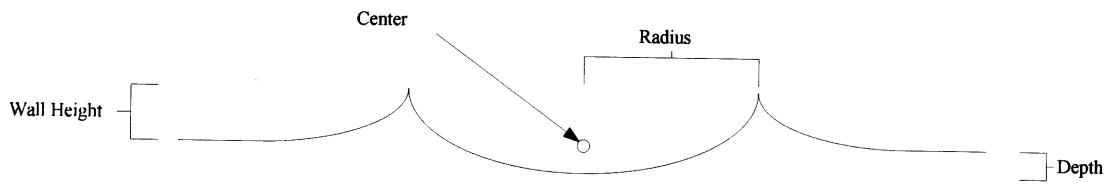

Figure 18: Crater attributes

- The **impact elevation angle** is the angle, measured from vertices y-axis and in degrees, that the meteor would have impacted the surface. For instance, an impact direction angle of 90° indicated that the meteor would have impacted the crater from directly above. The range of acceptable values is 0° to 90°.

- The **impact direction angle** is the angle, measured in the x and z axes and in degrees, that the meteor would have impacted the surface.

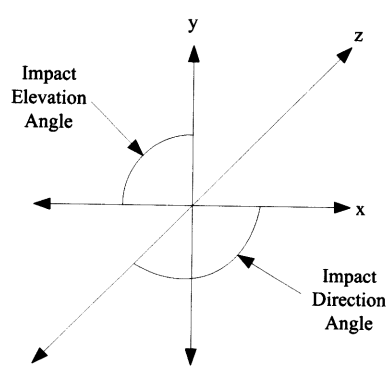

Figure 19: Impact elevation and direction angles

After the user specifies the attributes of the crater, the crater is created. For each crater, a generic profile curve from the center of the crater to the crater wall is assumed. From the crater wall outward, two different profile curves are supported. The first follows an exponential decay, while

the second follows a linear decay. The curve ultimately goes to 0. When the two curves are combined, they form a piece-wise continuous curve. Once the profile curves of the crater are defined, they are incrementally transformed 360° around the vertical creating a surface of revolution.

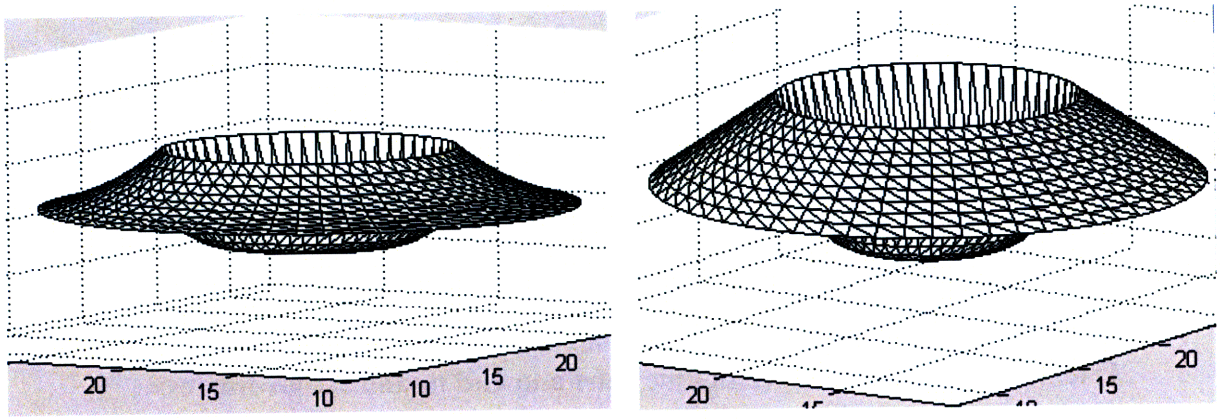

Figure 20: Crater with exponential decay(left) and linear decay (right)

4.3.2 Control Points Features

The goal with the control points features is to allow the user to place control points on the surface that can be used to "shape" the mesh. For instance, the user can use control points to create the following terrain features:

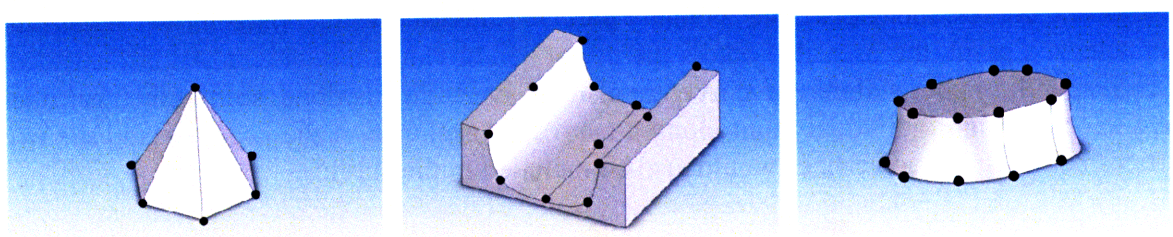

Figure 21: Examples of a mountain, valley, and plateau and their respective control points

One important note is that the user is not limited to creating a single feature in the control points feature section. Within one feature, the user could place enough points to create a mountain, valley, and plateau.

The control points feature has the following attributes:

- The **longitude and latitude range** are the span of values that the control points feature spanned and are measured in degrees.
- The **resolution** is the distance between the vertices in the section and is measured in kilometers.
- The **control points** are the points on the surface that the user assigned some initial weight and height to. These are the points that influence the others when the surface grows. Each control point has the following attributes:

- **Longitude**
- **Latitude**
- **Altitude**
- **Weight**
- **Initial Height**

- The **number of iterations** allows the user to pick the number of times to let the surface grow.

- The **texture** is a texture to overlay on the surface. This is used, so that the user can place control points relative to features on the texture in hopes of creating a 3D version of the 2D texture. The user also specifies the **start and end u- and v-coordinates**.

When constructing the control-points feature, the user has two means of adding control points to the surface: through picking individual points or constructing curves of points.

When selecting individual points, the user picks the location on the mesh and then enters the initial height value and weight for that point. Once a point is chosen, interpolation is used to find the nearest point in the mesh grid.

When selecting curves, the user selects as many points as the user wants on the curve. Whenever the user selects a point, the user also enters the initial height value and weight. Additionally, all of the corresponding points along the curve are also selected. This is done, so that the user does not have to select each individual point along a curve. For instance, if the user selected the two red vertices below, the blue vertices would also be included in the curve.

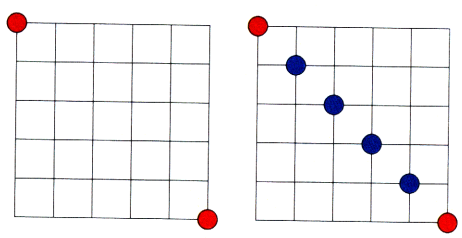

Figure 22: Example of interpolation when constructing curves

For the points that are added to the curve, the initial heights and weights of the surrounding vertices are used to determine their height and weight. The number of vertices between the two is determined and the change in weight and height values is divided by this value to get the increment for each vertex.

Once the control points are selected, they are then used to generate the surface. For each vertex in the mesh, the effects of the control points are determined. For each control point, the distance from the vertex to the control point is calculated. The weight and height of each control point is then scaled by the inverse of the distance and added to the height and weight of the vertex. The

goal with scaling by the inverse of the distance is to have the control points more heavily influence the vertices that they are near.

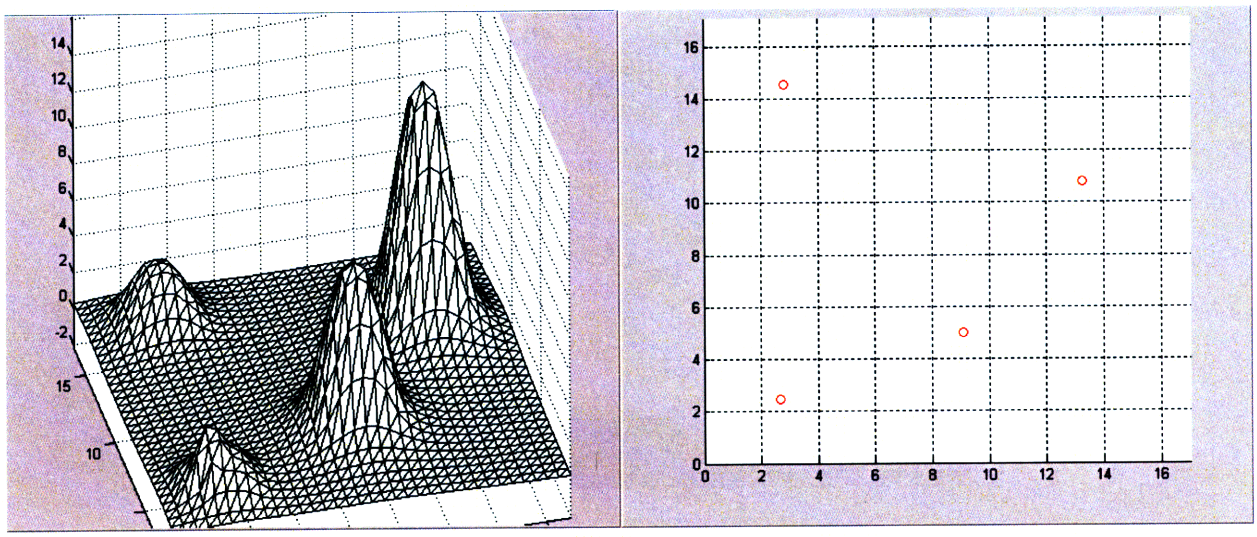

Figure 23: Control points used to construct four mountains

4.3.3 Boulders

A boulder has the following attributes:

- The **center** is the longitude and latitude coordinates of the center of the crater and is measured in degrees. The center is assumed to be located on the surface of the planet, and therefore, the altitude is always set to the radius of the planet.

- The **radius** is the distance from the center of the crater to the edge of the crater wall and is measured in kilometers.

Since there are a wide variety of shapes that a boulder can take, two pre-defined shapes are used for boulders: sphere and cube. For the sphere, the radius is the radius of the sphere. For the cube, the radius corresponds to the length of one side.

4.3.4 Distribution Curves

The goal of the distribution curve is to randomly generate either craters or boulders with varying sizes and locations and apply them to the terrain. The distribution curve has the following attributes:

- The **type** indicated if the distribution curve is for boulders or craters.

- The **longitude range** and **latitude range** indicate the span of the distribution curve.

- The **start radius** and **end radius** indicate the range of radius values that the features could take on.

- The **density** indicates the number of features per square kilometer.

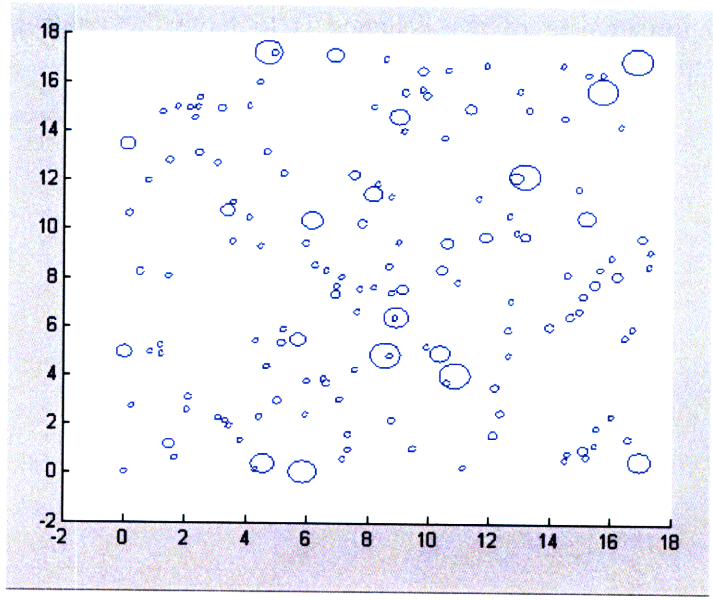

Figure 24: Example of boulder distribution

The code first calculates the number of features needed based on the density and the longitude and latitude ranges. It then loops through each feature and assigns a random location and radius. While the location is random, the radius follows a chi-squared distribution; this means that the majority of the features will have a radius near the minimum limit, while few will have a radius near the upper limit.

If the user chooses craters as the features, then the wall height and crater depth are assigned random values, while the impact direction and elevation are assumed to be from straight above. The decay type is assumed to be linear as well. Once the craters have been generated, then they are added to the terrain in the same way that an individual craters is.

4.3.5 Randomness

Randomness can be added to individual features. When adding it to the individual features, the user specifies a random seed and the number of iterations. During the iterations, a random number between 0 and 1 is generated and then scaled based on the size of the feature. The height value is changed by this amount.

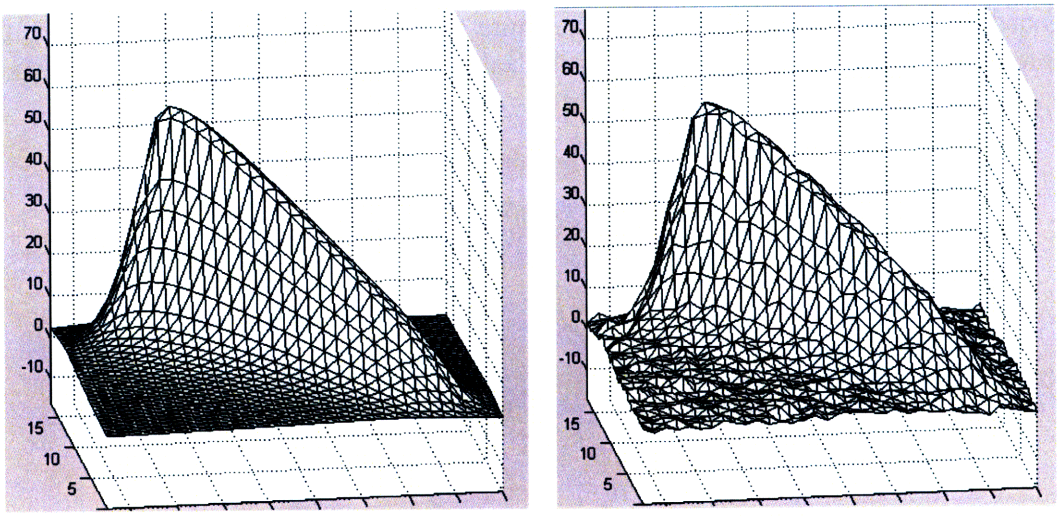

Figure 25: Control points feature with and without roughness

When the user adds randomness, three copies of the height values of the feature are kept: the original feature, the current height values, and the previous height values. This allows the user to restore the baseline feature or undo the last iteration of randomness.

4.3.6 Adding and Removing Features

When craters or control points features are created, they are added to the mesh of the planetary surface. To do so, the extents of the mesh grid locations, which are overlapped by the feature's grid, are determined. The surface map of the feature uses a bi-directional interpolation to determine the feature surface height at each of the corresponding terrain grid locations. The height values of the feature at each terrain grid point are then added to the terrain's current elevation.

The height values are stored separately for each feature, so that if the feature needed to be removed, it could be subtracted from the planetary surface without redoing the interpolation of the points. The tradeoff between storing the extra data and re-doing the interpolation is made because it eliminated the need to redo the interpolation and eliminated any rounding differences that may have occurred as part of the interpolations.

When the user creates boulders, the boulders are not displayed on the overall terrain mesh. This is done because the boulders are often much smaller than the resolution of the mesh and drawing a large number of boulders negatively impacted the performance of the tool. Thus, the boulders are drawn only in the geomorphing application.

4.4 Textures

The user also has the ability to apply texture maps to the surface. The textures are used to determine the color values of the vertices. The following formats are supported: Tagged Image File Format (TIFF), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG) formats.

When applying textures, the user can either apply a single texture to the overall surface or apply individual textures to each tile. For each texture, the user specifies the filename and the start and

end u-coordinates and v-coordinates. To determine the texture value for a vertex, the u and v-coordinates for a vertex are found, and then the value in the texture map corresponding to those coordinates is assigned to the color of the vertex.

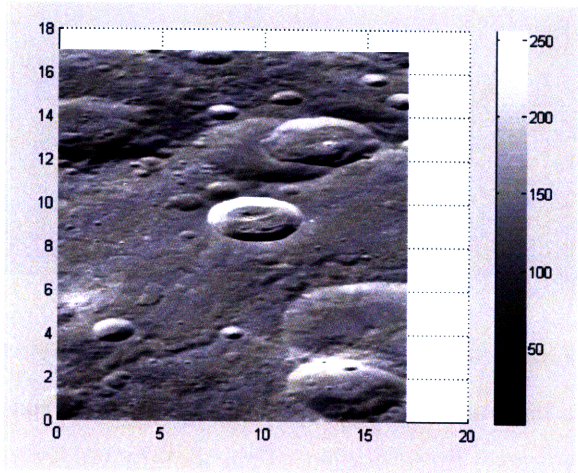

Figure 26: Sample texture

4.5 Lighting

4.5.1 Overview

When applying lighting to the surface, the user has two options: apply a light map or create an individual light source. If the user chooses to place a light source, then it is ultimately used to generate a light map for the surface. Light maps are used in both cases because it would be expensive to calculate the light value at each vertex during the simulation; as the vertices morph, their normals change, which influences the light value. Instead, the light values are assigned using the light map. Using a light map fixes the lighting conditions while the terrain morphs. While this method is not exact, it provides a good approximation for the actual light values. If the user does not specify a light source, then infinite lighting is assumed.

4.5.2 Light Sources

When creating a planetary surface, the user can place a light source. A light source has the following attributes:

- Latitude
- Longitude

The light source is assumed to be white light with full intensity; its distance is also assumed to be at infinity. When creating a light source, the user also then specifies the red, green, and blue ambient and diffuse components of the surface.

To determine the contribution of light at each point, ray tracing is used. Ray tracing typically applies to lighting in that rays are shot from the viewpoint to the objects in the scene to determine which are in the light and which are in shadows. For these simulations, the situation is different in that the viewpoint of the user is not known beforehand. Therefore, for each vertex, a

ray must be cast from it to the light source to determine if it is visible from the light source and if it is in the shadow of another object. The color of the vertex is the sum of the diffuse and ambient components of the material. If the vertex is in the shadow, then only the ambient color is used. If the vertex is not in the shadow, then it is multiplied by the dot product of the vertex normal and the direction to the light source.

The light map is created when the terrain database is generated, using the built-in functionality for generating images within Matlab.

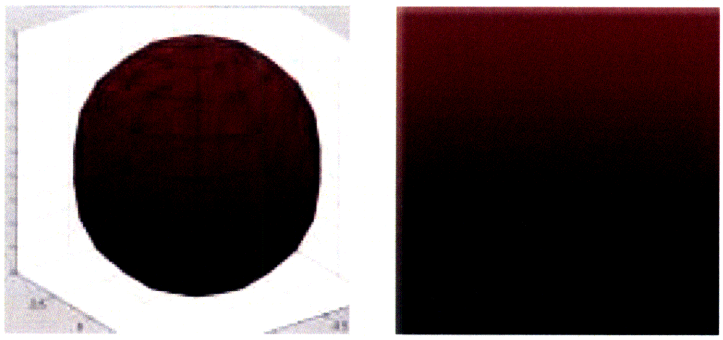

Figure 27: Surface and corresponding light map

4.5.3 Light Maps

The user also has the ability to apply a light map to the surface. The following formats are supported: Tagged Image File Format (TIFF), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG) formats.

When applying a light map, the user selects the light map to apply to the entire surface. The user specifies the filename and the start and end u-coordinates and v-coordinates. To determine the light map value of the vertex, its color value is assigned from the u and v-coordinates in the light map texture.

4.6 Terrain Database Generation

After the user completes creating the planetary surface, the terrain database is generated. The terrain database contains all of the information that the geomorphing application will use to build the terrain. When the terrain database is generated, only the final terrain mesh is stored; the information about the individual features is not relevant to the simulation.

The levels-of-detail used in the geomorphing application are generated from the finest terrain mesh. The number of levels-of-detail is determined by the value n that the user specifies where the number of levels-of-detail equals n + 1. This is because the smallest number of vertices a level-of-detail can contain is two (i.e. $2^n + 1$ where n is 0). For example, if n equals 2 initially, then the first level-of-detail has 5 vertices (n = 2), the second has 3 (n = 1), and the third has 2 (n = 0). The levels-of-detail are found by removing every other vertex from the previous, finer level-of-detail.

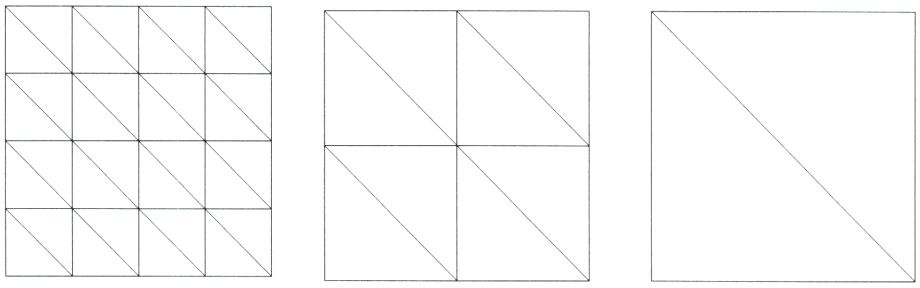

Figure 28: Three levels-of-detail

The following illustrates the 1st, 4th, and 6th levels-of-detail for a 300 by 300 kilometers piece of Mars data.

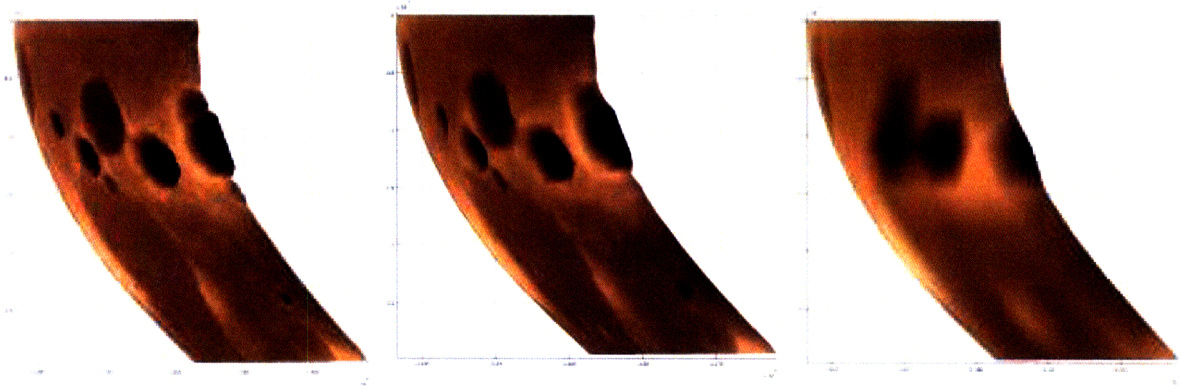

Figure 29: 1st, 4th, and 6th levels-of-detail with resolution 2.3 km, 18.4 km, and 73.6 km, respectively

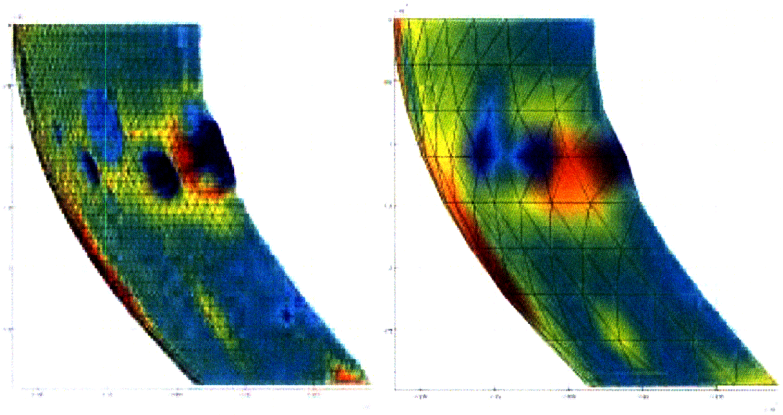

Figure 30: Triangles for the 4th and 6th levels-of-detail

For each level-of-detail for each tile, the user specifies the active range and the percentage of the active range when the vertices should morph. The active range corresponds to when the level-of-detail will be rendered. If the distance from the tile to the camera falls within the active range, then the level-of-detail is rendered. The morphing range is entered as a percentage and indicates during what percentage of the active range the vertices should morph. For instance, assume there are three levels-of-detail. The active range is 0 – 10 meters for the first level-of-detail, 10 – 20 meters for the second, and 20 – 30 meters for the third. Also, assume that the morphing

percentage is 50% for the first, 30% for the second, and 20% for the third. This would yield the following:

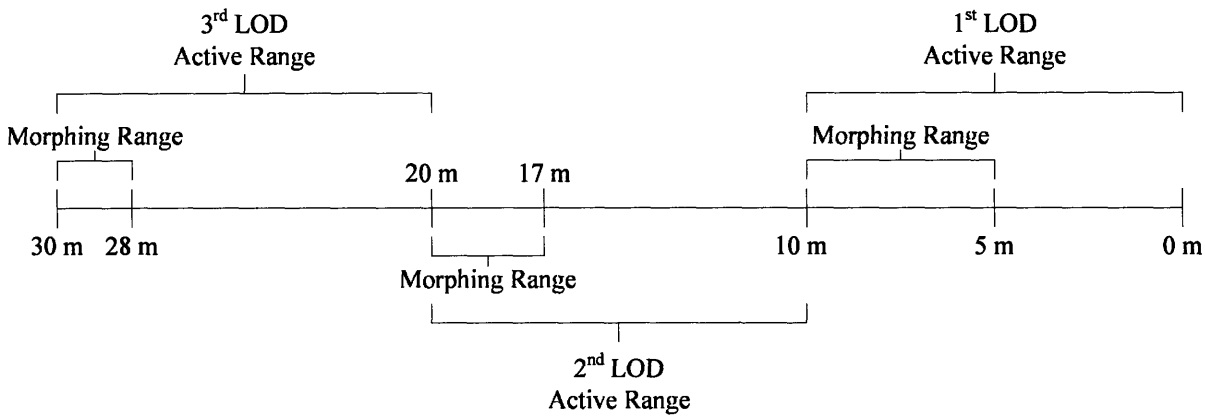

Figure 31: Active and morphing ranges

After the levels-of-detail are generated and the user specifies the active and morphing ranges, the following information is written to a binary file:

- For each tile
    - The tile indices of the left, right, top, and bottom neighbors of the tile if they exist
        - For each level-of-detail
            - For each vertex
                - X-coordinate
                - Y-coordinate
                - Z-coordinate
            - Active Range
                - Start
                - End
            - Morphing Percentage
        - Texture Information
            - Filename and start and end u-coordinates and v-coordinates of the texture
            - Includes tile coordinates if not one overall texture

- Lighting Information
    - Filename and start and end u-coordinates and v-coordinates of the texture

5 Geomorphing Tool

5.1 Pre-Processing

5.1.1 Geometry Creation

After the terrain database is generated using the terrain generation tool, the geomorphing tool loads the database and uses the information in the file to create the tiles, levels-of-detail, and vertices for the mesh.

The application iterates through the list of tiles and creates the faces for each level-of-detail for the tile. The faces are created using the vertices specified in the terrain database. When creating the faces, triangular meshes are used instead of individual triangles. This approach is chosen because the triangular strips are faster to process on the GPU.

Each tile contains three types of faces: inner faces, outer faces, and corner faces. Inner faces are faces that never touch a neighboring tile, while outer and corner faces may touch a neighboring tile. There is only set of inner faces stored because they never touch another tile, and therefore do not have to change.

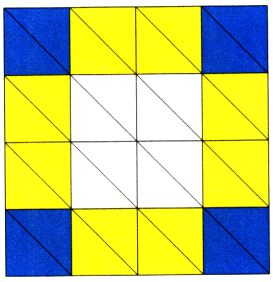

Figure 32: Corner faces (blue), outer faces (yellow), and inner faces (white)

The outer and corner faces are further broken up depending on their location in the tile. The outer faces are classified as left, right, top, and bottom. The corner faces are classified as top-right, top-left, bottom-right, and bottom-left.

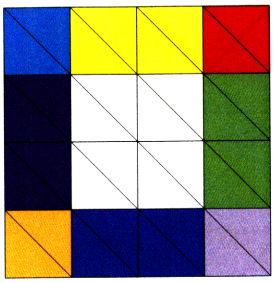

Figure 33: Triangular strips (each color indicates a different strip)

For the subtypes of the outer and corner faces, there are three different versions stored; the versions correspond to if the neighbor's tessellation level is the same, coarser, or finer as the tile's. Thus, these faces are used to eliminate the t-cracks that form when joining tiles with different tessellation levels.

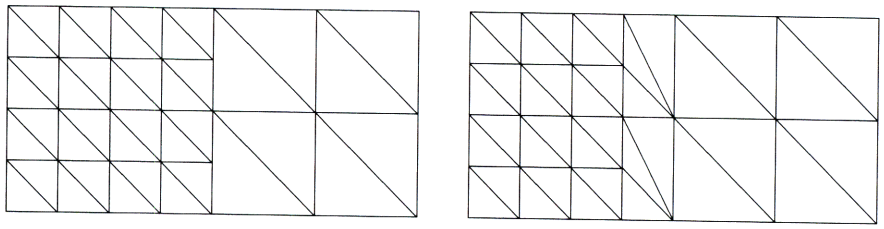

Figure 34: Inserting geometry to eliminate gaps

Storing the faces separately allows the application to render the appropriate faces based on the tile's tessellation level and its neighbors' tessellation levels.

Each level-of-detail is stored as a switch in OSGL, while each face type (left outer, top outer, top-left corner, top-right corner, etc.) is also stored as a switch. This is done, so that during the simulation, the application can set which level-of-detail and which faces should be visible and rendered; the other faces are set to invisible and not rendered.

After the geometry is created, the morphing values are calculated and set for each vertex. When morphing, there are four possibilities for the vertices (illustrated below) [Wagner]:

1. Vertex is at an odd x-position and even y-position: vertex has to move to the middle position between the next left ('1') and right ('2') vertices

2. Vertex is at an odd x-position and odd y-position: vertex has to move to the middle position between the next top-left ('1') and bottom-right ('3') vertices

3. Vertex is at an even x-position and odd y-position: vertex has to move to the middle position between the next top ('2') and bottom ('3) vertices

4. Vertex is at an even x-position and even y-position: vertex remains the same.

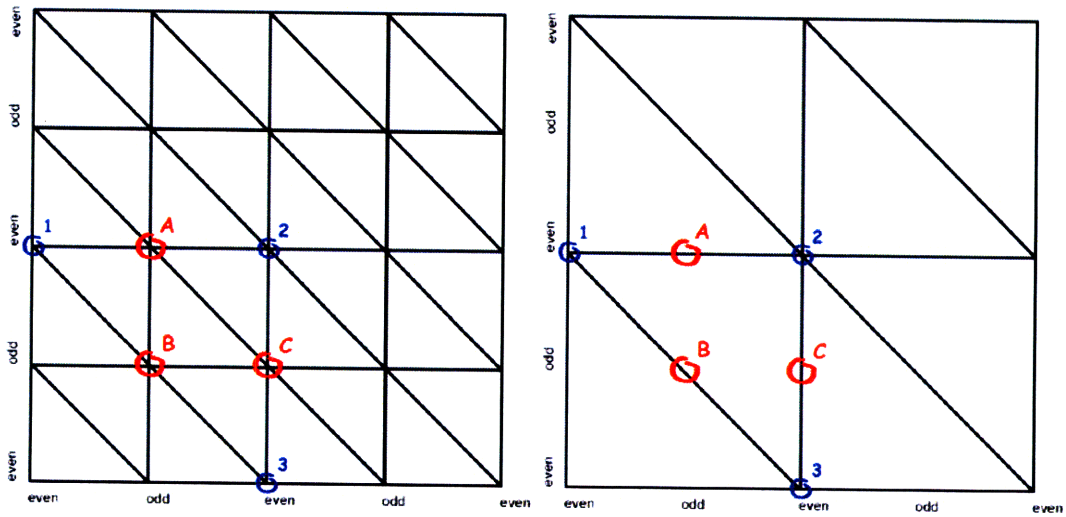

Figure 35: Possibilities for vertices geomorphing

The amount that a vertex must morph is determined using the middle position it has to move to. To determine the amount to morph, the average height values of the two neighboring vertices are

calculated. This is then subtracted from the height of the vertex to be morphed. This final value indicates the total amount that the height of the vertex will have to change when switching levels-of-detail. If the vertex does not morph, the value is set to zero.

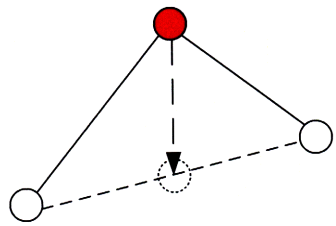

Figure 36: Vertex (red) morphing to position in coarser level (dotted circle)

After the geometry and morphing values are created, the boulders are created using their locations and radii. For each boulder, either a sphere or cube object is created in OSG. The boulders are stored separately from the faces and always rendered in the simulation.

5.1.2 Shaders

A shader refers to code written in the OpenGL Shading Language that can be executed on one of the OpenGL programmable processors. Currently, there are two types of shaders that can be created: vertex and fragment. After the shaders are written, they are compiles and linked to form executable code called a shader program. This program is loaded onto the graphics board for execution simultaneously with the OSG application.

When creating a shader, the application developer can define three qualifiers (attribute, uniform, and varying) that specify the type of input or output a variable is. An attribute variable is used to communicate frequently changing values from the application to the shader. The value in the attribute variable can be different for each vertex. A uniform variable is used to communicate infrequently changing values from the application to the shader. The value in the uniform variable is the same for each vertex. A varying variable communicates interpolated values from the vertex shader to the fragment shader. The value in the varying variable can be different for each vertex. In addition to these types, the developer also specifies the data type (e.g. int, float).

While it is possible to create multiple vertex and fragment shaders for an application where vertices use different shaders, one vertex shader and one fragment shader are used in the geomorphing application. This is done because the logic is the same for all vertices, so different versions were not needed.

There are two challenges when writing the shaders. First, the shaders overwrite the rendering functionality that is inherently built-in to the GPU. Therefore, any built-in functionality must be rewritten when using user-defined shaders. Second, the number of variables that the developer can define depends on the graphics card used. Unlike defining variables in an application, the hardware has a limited number of slots available for storing variables.

5.1.2.1 Vertex Shader Object

Vertex processing involves the operations that occur at each vertex. Some of these operations include: vertex transformation, normal transformation and normalization, texture coordinate generation and transformation, and lighting.

In the geomorphing application, the vertex shader is leveraged to morph the height values of the vertices. To do so, the vertex shader uses four, user-defined variables:

- The **height value difference** stores the value that the vertex has to morph. This is an attribute variable.
- The **tile percentage change values** stores the percentage that the vertices of each tile must morph. This is an array of values and of type uniform.
- The **tile index** stores which tile the vertex belongs to. This is an attribute variable. This is used to look up what percentage a vertex should change in the tile percentage change values array.
- The **texture coordinates** store the u and v coordinates of the vertex for the lookup in the texture.

During the simulation, the vertex shader first looks up the height value difference for the vertex. If it is 0, then there is no geomorphing, and it returns. If not, it multiplies the height value difference and the percentage to change (the percentage to change is 0 if the tile is not in the morphing range) to determine the new height of the vertex. The height value of the vertex is then updated with this new value.

5.1.2.2 Fragment Shader Object

A fragment refers to the information needed to render a pixel on the computer screen. Fragment processing consists of the operations that occur on a per-fragment basis, most notably reading from texture memory and applying the texture values at each fragment. Fragment processors are also responsible for: operations on interpolated values, texture access, texture application, fog, color sum, texture blending, light map blending, and calculating vertex color.

In the geomorphing application, the fragment shader is used for applying textures and light maps. To do so, the fragment shader uses five variables for geomorphing:

- The **texture present** variable indicates whether or not a texture is present for the vertex. This is a uniform variable.
- The **light map present** variable indicates whether or not a light map is present for the vertex. This is a uniform variable.
- The **texture map** variable stores the name of the texture map. This is a sampler2D variable.
- The **light map** variable stores the name of the light map. This is a sampler2D variable.

- The **texture coordinates** store the u and v coordinates of the vertex. This is a varying variable.

The fragment shader checks to see if a texture present. If so, then it sets the color of the vertex to the value in the texture. It then checks to see if a light map is present. If so, it multiplies the color by the value in the light map. It then sets the final color of the fragment, which is drawn on the screen.

5.2 Simulation

5.2.1 Supported Simulation Types

The tool currently supports two different types of simulations. First, the user can specify a trajectory file. The trajectory file contains the longitude, latitude, altitude, and orientation of the camera as well as the time step corresponding to each point. Second, the user can manually navigate around the surface, choosing to zoom in, zoom out, etc.

5.2.2 Geomorphing Logic

After the preprocessing steps have completed, the simulation begins. To start, the initial position of the camera is supplied, which is taken from either the trajectory or where the user has placed the camera.

At each instance, the current position of the camera is queried. For each tile, the distance from the camera's position to the center of the tile is calculated. This distance is used to determine which level-of-detail to use and whether or not geomorphing should occur. To determine which level-of-detail to use, the application checks which level-of-detail's active range contains the distance away. When choosing the levels-of-detail, there is a requirement that there not be a level-of-detail difference greater than one between two neighboring tiles. Consequently, once the levels-of-detail are determined, the application must go through and check to see if this constraint is satisfied. If it is not, then the level-of-detail used may be changed to satisfy the constraint, as below with the yellow tiles.

3	3	3	3	3
3	3	2	2	3
3	3	1	2	3
3	3	3	2	3
3	3	3	3	3

3	3	3	3	3
3	2	2	2	3
3	2	1	2	3
3	2	2	2	3
3	3	3	3	3

Figure 37: Adjusting levels-of-detail

Once the appropriate levels-of-detail are chosen, the application then determines which tiles (if any) have neighbors with different levels-of-detail. Depending on the tessellation levels of the neighbors, the appropriate outer and corner faces are selected.

After the geometry is selected, the application determines whether or not the distance from the camera point to the center of the tile falls within the morphing range of the vertex. If it is, it calculates the percentage the height value should morph. It then sets the uniform variable for each tile to indicate what percentage to morph, with zero indicating that the vertices in the tile do not need to morph.

On the next frame of the application, a new position is supplied to the camera, and the loop repeats itself until either the end of the trajectory is reached or the user chooses to exit the simulation.

5.2.3 Capturing Images

When running a simulation, the user has the ability to capture the images displayed during the simulation. If the user chooses to capture them, before the simulation begins, the user must supply the frame rate at which to take the camera snapshots. At each instance, the pixels of the viewing frustum's near clipping plane are captured and written to memory. After the simulation completes, the images can either be written to files or sent to another program via Ethernet.

6 Testing

6.1 Methodology

The hypothesis tested was that VBNS would perform better when geomorphing was used in simulations as opposed to just switching between different levels-of-detail without geomorphing. The assumption is that, when a level-of-detail is switched without geomorphing, there will be a significant change in the surface geometry, causing the line-of-sight (distance from the camera's location to the surface) calculation from the point in the trajectory to the surface to undergo a large correction. Whereas, when a level-of-detail is switched with geomorphing, there is a more gradual change in the surface geometry, which causes the line-of-sight calculation to undergo a much smaller, almost undetectable, correction.

When calculating the line-of-sight distance, the distance from the camera's position to the location of the pixel in the center of the simulation window is determined. Another approach considered is to find the intersection of the line-of-sight with the geometry and then calculate the distance between the camera's position and the intersection point. This approach was discarded because it did not detect the height-value changes that occurred in the vertex shader; the intersection point is calculated in the graphics pipeline before the vertices are morphed. The pixel is used instead because its location is determined at the end of the graphics pipeline after its location has been morphed.

To conduct the tests, a planetary model was built, which was 0.5 degrees by 0.5 degrees, had a resolution of 100 meters, and had a radius similar to the Moon. When constructing the surface, a wire mesh frame with a solid fill was used, excluding lighting and textures because they did not have any effect on the line-of-sight calculations.

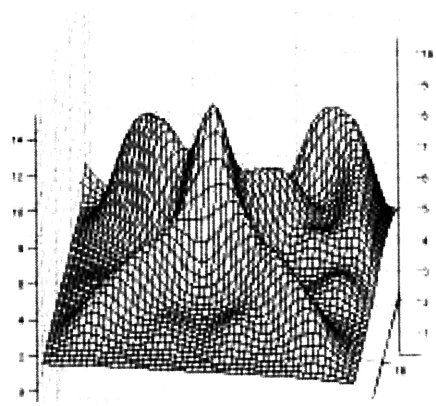

Figure 38: Test surface

A trajectory was also created that was used to fly over the surface. It started at a height of 50 kilometers, traveled across the surface, and then ended at a height of 1 kilometer. The length of the trajectory was 60 seconds. The trajectory included the line-of-sight angle, which was chosen to always point straight down. This line-of-sight angle was selected to minimize discontinuities that could arise when an angular one was used:

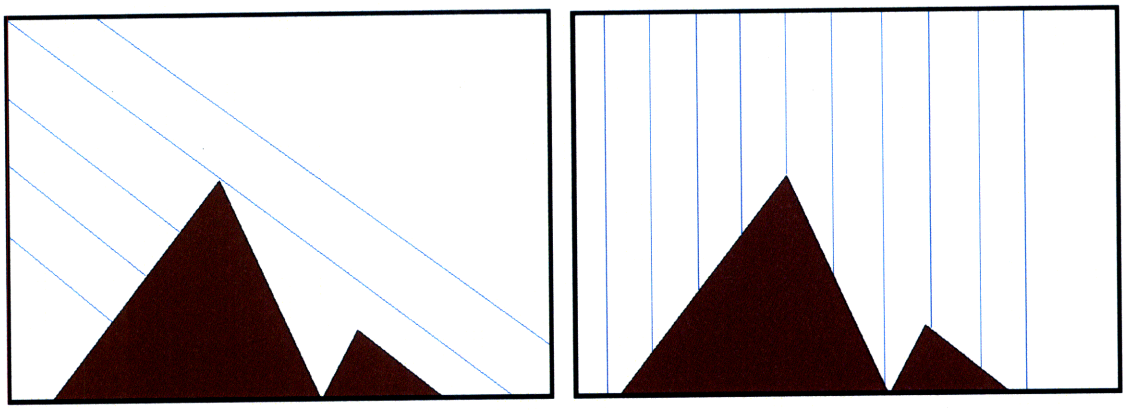

Figure X: Discontinuities occur when the line-of-sight (in blue) travels over the leftmost peak

The first test needed was running the simulation over the surface with only one level-of-detail and no geomorphing. The results of this would indicate what the actual line-of-sight value should be at each point of the trajectory and would provide a baseline to compare the other tests with. The next test conducted involved running the simulation over the surface with only switching the levels-of-detail. The final test conducted entailed running the simulation over the surface with geomorphing. Comparing the line-of-sight calculations for each would indicate how the geomorphing performed.

All of these tests were conducted on a machine with the following specifications:

- 64-bit Linux Extended Memory Technology operating system
- Dual core processors (3.75 gigahertz Xenon processors)
- 16 gigabytes of memory
- NVIDIA Quadro FX 4500 Graphics Board, which had the following attributes:
    - 512 megabytes of memory
    - 181 million triangles per second
    - OpenGL version 2.0
    - Shader Model version 3.0

6.2 Results

The results of one of the simulations are presented in figure 39. The red line indicates the line-of-sight calculations for when only a single level-of-detail is used. This shows the actual distance between the camera's position and the surface at each time instant. The green line shows the line-of sight calculations when levels-of-detail are used without geomorphing, while the blue line shows the line-of-sight calculations when geomorphing is used.

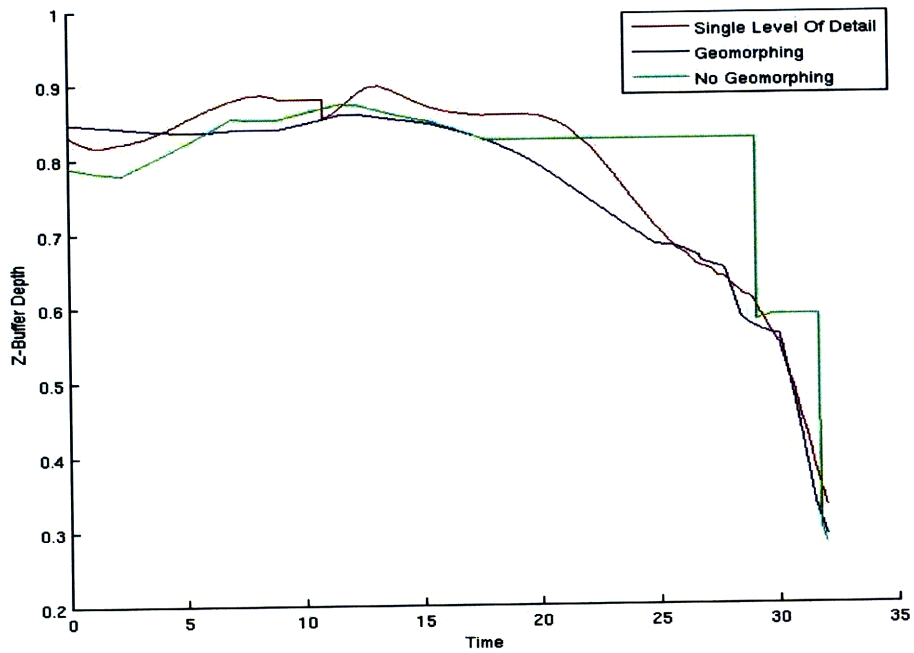

Figure 39: Results

When looking at the results of switching the levels-of-detail without geomorphing, there are two portions of the line where there are significant drops in the calculated distances. These occur where the levels-of-detail are switched and indicate that there is a large correction that occurs in the calculation when the levels-of-detail are switched. When looking at the results of switching the levels-of-detail with geomorphing, there are no significant drops when the levels-of-detail are switched. These results support the hypothesis that switching levels-of-detail with geomorphing would result in a more gradual change in the geometry than switching levels-of-detail without geomorphing.

When looking at the levels-of-detail with and without geomorphing, neither line follows exactly the same path of when a single level-of-detail is used. The discrepancy between the lines is much larger in the beginning of the simulation than the end. This is because the coarsest level-of-detail is used in the beginning of the simulation, whereas the finest level-of-detail is used in the end of the simulation. This indicates that using levels-of-detail, other than the finest, results in the loss of some of the information about the planetary surface. While there was some loss of information, the number of triangles that are rendered in the beginning of the simulation is significantly less than when using a single level-of-detail (64 vs. 512, respectively).

This page intentionally left blank.

7 Future Research

One of the challenges when building the planetary models for the geomorphing simulations is to determine how many levels-of-detail to generate. When using one level-of-detail, the finest geometry is used in the simulation. While this provides the most accurate model of the surface, it also means the most amount of geometry is drawn. As more levels-of-detail are used, less geometry is drawn, but some of the information about the planetary is lost when the coarser levels-of-detail are used. It would be useful to investigate the trade-offs between the levels-of-detail and the information lost.

In the terrain generation tool, if a light source is specified, a light map is generated. Since the viewpoint of the simulation is unknown, rays must be cast from each vertex to the light source to determine whether or not is in the shadow of another object, hidden from the light source, etc. As the mesh size increases, computing the light map becomes more computationally expensive and extremely slow to compute. It would be useful to investigate other ways of generating the light map to be used in the geomorphing simulations to reduce the amount of time required to create it.

In the geomorphing simulations, a light map is used to approximate the light values of each vertex since the normal changes as the vertex morphs. Although the light map provides a reasonable approximation of the light value, it would be useful to investigate ways to recalculate the light value in real-time when the vertex morphs, so that the light values are more accurate.

In the geomorphing simulations, the boulders are currently rendered at all times because they are not associated with a level-of-detail. It is assumed that, when a vision-based navigation system is far from a planet, it cannot detect all of the boulders. It would be useful to determine when these objects should and should not be rendered in a simulation as well as if there is a way to approximate them using a similar method to levels-of-detail.

Finally, the geomorphing should be tested using the VBNS navigation and recognition algorithms. While the line-of-sight calculation provided a reasonable way to view the disruptions that switching the level-of-detail cause with and without geomorphing, the simulations need to be tested with the VBNS to determine how well they work. This will also indicate if there are any constraints on the size of the resolution that should be used, how many levels-of-detail should be generated, etc.

This page intentionally left blank.

8 Conclusion

Presently, the models that the vision-based navigation systems use in their simulations have several problems. The planetary data has a coarse resolution, sections with incomplete information, and inconsistent lighting conditions. Data is also only available from the Moon and Mars, limiting the types of terrains used in the simulations and affecting the testing of the VBNS. Thus, the purpose of this research was to develop the three-dimensional models of planetary surfaces which can be used in the test environments for the VBNS algorithms.

The terrain generation tool was created which allows users to either create entirely synthetic planetary surfaces or manipulate existing planetary data. Users can create craters, mountains, boulders, valleys, etc. and place them on the planetary surface. The users can also add textures and lighting to the surfaces, creating realistic surfaces. Since the user can manipulate existing data, the tool solves the problems with the existing planetary models that the VBNS use.

To render the large amount of data required with the planetary models, geomorphing was implemented and incorporated into the simulations. The test results indicate that switching levels-of-detail with geomorphing result in a more gradual change in the geometry than switching levels-of-detail without geomorphing. Using the levels-of-detail results in the loss of some information about the planetary surface, especially when the coarsest level-of-detail is used. However, it also reduces the amount of geometry that is rendered. The question becomes how much data can be lost when using the levels-of-detail without negatively affecting the vision-based navigation systems. To determine this, the feature recognition and navigation algorithms should be tested with the geomorphing to see how the loss of information affects the algorithms. Knowing this would help indicate how many levels-of-detail to use in future simulations.

Using the terrain generation tool and the geomorphing, the vision-based navigation systems can now be tested in a broader set of environments. The user can create a planetary surface with the necessary features and supply a trajectory to the simulation. The VBNS can then fly the trajectory over the planet, taking pictures along the flight and dynamically updating the imagery passed to the VBNS.

This page intentionally left blank.

Acknowledgements

First, I would like to thank **Bryan Wasileski**, my supervisor at Draper Laboratory. When I first began my research, it had been two years since I last programmed, and the field of graphics was relatively new to me. It would have been near impossible to complete my research without your constant mentoring and guidance. You were also instrumental in writing my thesis, providing critical feedback throughout the process.

The work with OpenSceneGraph would not have been possible without the help of **Justin Vican**, a software engineer in my group at Draper. Thank you for answering so many of my questions about OpenSceneGraph, explaining how to properly call the APIs and walking me through examples of your own work. On numerous occasions, he also helped me work through issues with my code.

I would like to thank **Professor Popović**, my thesis advisor, for your guidance throughout my research and feedback on my thesis.

I would like to thank my **mother** and **father** for their unwavering support and encouragement, not just in graduate school but throughout life. Leaving my position at Microsoft to return to graduate school at MIT was one of the most difficult decisions I have ever faced, and you helped me through it. To my dad, we have climbed many mountains together, and I look forward to climbing many more in the years to come. To my mom, you are the most selfless and loving person I know.

Finally, I would like to gratefully acknowledge the financial support for this research, which was provided by **Draper Laboratory**. Without it, this research would not have been possible.

This page intentionally left blank.

References

Asirvatham, Arul, and Hugues Hoppe. Terrain Rendering Using GPU-Based Geometry Clipmaps. *GPU Gems 2.* Addison-Wesley, March 2005.

de Boer, William. *Fast Terrain Rendeing Using Geometrical MipMapping.* E-mersion Project, October 2000. http://www.connectii.net/emersion.

Ferraris, Jonathan. *Quadtrees.* http://www.gamedev.net/reference/programming/features/quadtrees/.

Hidden Surface Determination. *Wikipedia: The Free Encyclopedia.* January 20th, 2008.

Horne, Richard. *3DEM Software for Terrain Visualization and Flyby Animation.* http://www.visualizationsoftware.com/3dem.html.

Larsen, Bent, and Niels Christensen. "Real-time Terrain Rendering using Smooth Hardware Optimized Level-of-detail." *Journal of WSCG, Vol. 11, No. 1, ISSN 1213-6972.* 2003.

Losasso, Frank, and Hugues Hoppe. "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids." *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)* 23(3).

Luebke, David, et. al. *Level-of-detail for 3d Graphics.* San Francisco: Morgan Kaufmann Publishers, 2003.

Myrtz, Paul. *Generating Random Fractal Terrain.* http://www.gameprogrammer.com/fractal.html.

Rost, Randi. *OpenGL Shading Language.* Boston: Addison-Wesley Professional. 2006.

Schneider, Jens, and Rudiger Westermann. "GPU-Friendly High-Quality Terrain Rendering." *Journal of WSCG, Vol. 14, No. 1, ISSN 1213-6972.* 2006.

Sutherland, Ivan, Robert Sproull, and Robert Schumacker. "A Characterization of Ten Hidden-Surface Algorithms." *Computing Surverys, Vol. 6, No 1.* 1974.

"Vertex Shader." *Wikipedia: The Free Encyclopedia.* December 8th, 2006.

Vlietninck, Jan. *Hardware trilinear displacement mapping without tesselator and vertex texturing.* http://users.belgacom.net/xvox/technical.htm.

Wagner, David. "Terrain Geomorphing in the Vertex Shader." *ShaderX^2 – Shader Programming Tips and Tricks.* Woodware 2003.