University of Nevada
Reno

Out-of-Core Data Management for Planetary Terrain

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

by

Cody J. White

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

August 2011

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

CODY JAMES WHITE

entitled

Out-Of-Core Data Caching For Planetary Terrain

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

MASTER OF SCIENCE

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

Sergiu Dascalu, Ph.D, Committee Member

Scott Bassett, D.Des, Graduate School Representative

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

August, 2011

Abstract

Rendering terrain on a planetary scale can quickly become a large problem. Aside from the challenges of rendering terrain over a spherical body, the amount of data that needs to be processed to accurately render such terrain can reach the terabytes and beyond. Most terrain renderers focus on a particular region of a planet and are therefore limited to only a very few datasets to generate a proper rendering of that area. However, since planets are made up of such large areas, a different approach needs to be taken in order to display high-detail terrain around a viewer while sorting through the large amounts of planetary data available. Additionally, since modern desktops have a relatively small amount of memory, a system to swap data from the hard drive into graphics processing unit (GPU) memory needs to be created. Therefore, we present a data caching mechanism for planetary terrain rendering which can efficiently swap only the data around a viewer into and out of GPU memory in real-time. In order to speedup the process, we utilize the multi-core processing power of the GPU to perform data composition for use by a terrain renderer. Using this method, the CPU is able to perform search operations for new datasets and swap out old datasets while the previous ones are being rendered by the system. Additionally, we present a method for adding new datasets at runtime using the parallel processing abilities of the CPU. We achieve efficient framerates for high-quality views of terrain while minimizing the amount of time it takes to find data centered around a viewer and display it to the screen.

Acknowledgments

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

Thanks to my committee: Dr. Frederick C. Harris, Jr., Dr. Sergiu M. Dascalu, and Dr. Scott Basset.

Additionally, I'd like to make a special thank you to Joseph Mahsman because without him, this work would have never come to be. He helped me by providing interesting ideas and gave me a basis for my work. Also, thanks to Roger Hoang for his unparalleled technical expertise.

Overall, I'd like to thank my family for putting up with me for all this time and understanding when I had to get a lot of work done.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For many years, there has been a large amount of research pertaining to the rendering of realistic-terrain. This research spans the fields of video games, scientific visualization, and training simulations, just to name a few. However, there has been a very small amount of research that attempts to solve the problems of rendering full-scale planetary terrain in high-detail.

When rendering terrain on a planetary level, there are several things to consider, such as the shape of the planet and the datasets available which make up the planet surface. These datasets can range from being very small in size to several terabytes of information. Additionally, there can be hundreds to thousands of datasets which need to be used to accurately render the entire surface of a planet, making much of the research in terrain rendering inapplicable to addressing this problem.

Rendering terrain with large datasets has already largely been researched by many authors as it is the first step to accurately modeling realistic-terrain [7, 11, 15]. Typically, data is organized into some hierarchy and then chosen for rendering based on a user-specified search criteria. Therefore, only small amounts of the data are being processed at any one time, alleviating the need to process all the data at once. However, much of this work is focused on rendering a specific region of a planet and not the whole planet itself, using only one dataset at a time. While this research is extremely helpful in handling large datasets efficiently, it does not solve the problem of multiple datasets needing to be considered.

More than likely, planets will also be made up of a large amount of datasets which

are both big and small in terms of information. On top of this problem, the amount of graphics processing unit (GPU) memory available to desktop machines is not capable of storing these datasets at runtime. Therefore, only a few of the datasets exist in memory (in-core) at any one time, specifically the datasets which aid in rendering the visible terrain, while the rest are stored on the hard drive (out-of-core) until needed. The problem then, is how to determine what is and is not in view if the data is existing on the hard drive and not in the main system memory of the computer. The use of a spatial subdivision hierarchy can be used to alleviate this problem as datasets can be grouped based on their relative geographic locations. This hierarchy can then remain in memory without any data loaded simply to determine which datasets should be tested for rendering when the user makes a search query. As the number of datasets is large, creation of the hierarchy should happen in a preprocessing step and written to the hard drive, obviating the need for the runtime version of the algorithm to recreate it.

As a core requirement of terrain renderers, the algorithm should work in realtime. Therefore, the user must be able to move around the virtual world while seeing high-detail terrain at all times. To achieve this, the data caching mechanism should take advantage of both parallel processing on the CPU and offloading data to the GPU for later rendering. Much of the searching and swapping procedures can happen in parallel to the rendering algorithm so that the system is not slowed down by searching through the multitude of datasets. As well, the GPU can be utilized to perform composition of the terrain into a final image for the terrain renderer to use in generation of the three-dimensional mesh. Recently, the advances in graphics hardware have made joint CPU-GPU processing both available and easy through the use of programmable shaders which run directly on the GPU. Therefore, the GPU is an easy candidate to use for speeding up a potentially slow part of the algorithm.

In order to properly utilize the computing resources available, a level-of-detail (LOD) algorithm must also be devised in order to choose the correct LOD for a given dataset. As the viewer gets closer to a dataset, the LOD should increase and a more

detailed terrain mesh should be rendered. However, as the viewer gets farther away from a dataset, the LOD should decrease and terrain at a lower detail should be rendered. This approach is commonly used in terrain rendering algorithms in order to only create realistic terrain around the viewer and not waste time processing both occluded and distant terrain [5, 11, 17].

We present a method for taking large datasets and transforming them into manageable chunks as well as organizing the data into a user-searchable hierarchy that exists on the hard drive which can be used for any terrain renderer. As data is needed, it is swapped in-core and used in the creation of a group of textures which makes up the renderable terrain (a texture atlas). Once the user-defined maximum amount of memory has been reached, data which is no longer being rendered is discarded back to the hard disk for future search queries. As mentioned above, once a texture atlas has been created, it is uploaded to the GPU for further processing into the final image while the next search query is being performed. Additionally, our algorithm is capable of adding new datasets and using them for rendering at runtime.

In addition to the data caching of planetary datasets, we also present a method for selecting the proper LOD of a given dataset based upon the distance to the terrain of the viewer and the error introduced by the creation of a LOD hierarchy. Using this method, a terrain renderer does not need to know anything about the datasets which it is rendering in order for a continuous LOD to be performed and close datasets will have a high LOD while datasets farther away will have a lower one.

This work includes several contributions:

- We adapt common out-of-core data caching techniques [5, 8] for planetary scale-data. This is accomplished by placing datasets into volumes based on their geographic locations (Chapter 3).

- We adapt an error-based algorithm for a continuous LOD for heightmap data [17]. The error of a given piece of a dataset is computed as the difference of the maximum value of a data chunk and the maximum of the new averaged chunk for

different mipmapping levels (Section 3.3.2).

- The GPU is utilized to speedup terrain composition and allow for simultaneous searching and rendering. Additionally, multiple cores of the CPU are used to speedup the searching process (Section 3.3.3).

- New datasets can be created and rendered at run time. Our algorithm is capable of allowing the user to specify new datasets which can be fit into the planetary bounding volume hierarchy (BVH) for rendering (Section 3.3.4).

- Our algorithm is easily used in a virtual reality environment. We perform our GPU data uploads per context, allowing for multiple views of the same data to be rendered (Section 3.4).

Our algorithm efficiently swaps data into main memory and back out again based on the viewer's position in the virtual world (Chapter 3). This is performed by partitioning a dataset into tiles and organizing them into a quadtree where each level of the tree represents a different LOD for rendering. Each dataset is then organized into a planetary BVH which orders the datasets based on their geographic locations on the planet. For searching purposes, the geographic coordinates of the camera are supplied to the algorithm and it determines both what is visible as well as what LOD to use based on the distance of the viewer to the dataset and the user-definable maximum screen-error threshold. Once this data has been obtained, it is uploaded to the GPU for rendering and another search query can begin. The datasets can contain both height, color, and normal maps.

We implemented the algorithm as a data caching library linked with a terrain renderer in order to meet our requirements (Chapter 4) and found that the algorithm provided both fast frame rates as well as quick searches for finding the appropriate datasets to use for the rendering of high-quality terrain (Chapter 5).

Additionally, we present some ideas for how the research can be expanded to allow for data compression, optimal hard-drive layout, and addressing some visual caveats of the system (Chapter 6).

Chapter 2

Background

In order to generate realistic terrain, there must be an adequate number of datasets available to the terrain renderer. These datasets are typically stored in a texture format so that they can be uploaded to the GPU where standard texture mapping algorithms are applied. The most common type of dataset is the heightmap, which is a one-channel grey-scale image which stores the heights of the terrain covered by the image in each of its pixels, as shown in Figure 2.1. Using heightmaps, a terrain renderer is capable of offsetting the vertices of a polygon according to the heights defined by a heightmap, generating terrain defined by the resolution of the image. Once the mesh has been generated, any color and normal maps can be applied in order to produce the final image. As these datasets are stored as images, they can be trivially combined on the GPU for the final rendering.

A common approach to breaking down datasets into smaller, more memory cohesive chunks is to subdivide them with a quadtree hierarchy [5, 10, 19]. This type of structure makes sense because it splits its data into four equal-sized chunks per node until a predefined threshold has been met. Figure 2.2 illustrates quadtree subdivision of an image. Using this approach, datasets can be broken down and stored as pieces of the whole image so that only the parts that are needed can be used for rendering. Typically, the high-resolution imagery is stored in the leaf nodes of the tree and the parent nodes contain successively lower resolution versions of their four children. Therefore, different levels of the tree relate to a different LOD for the given dataset.

Many algorithms also use an adaptive LOD approach to rendering the terrain [12,

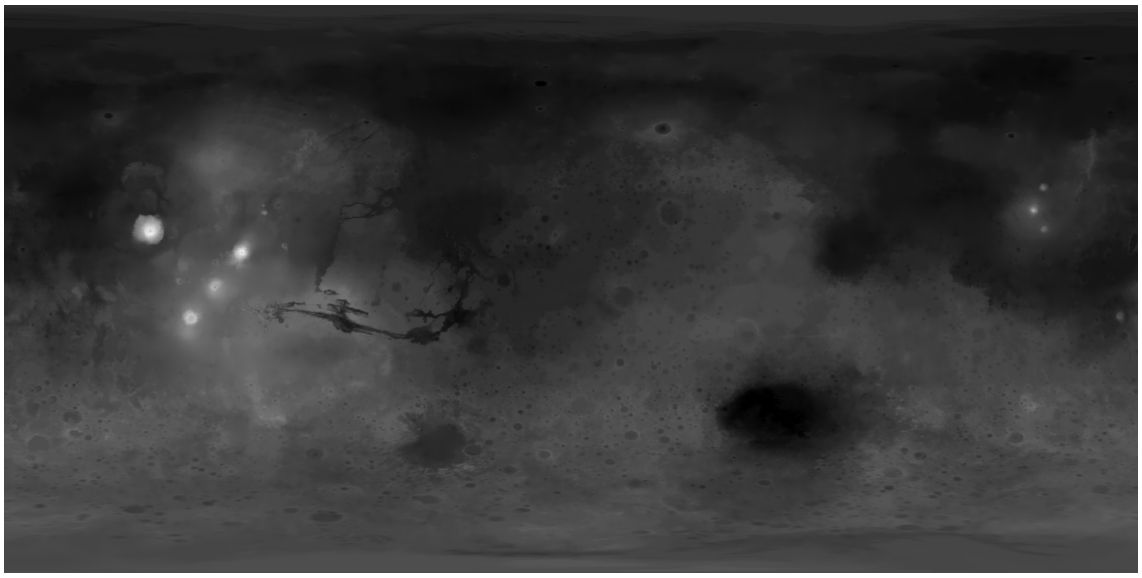

Figure 2.1: A heightmap generated from Viking imagery of Mars [22].

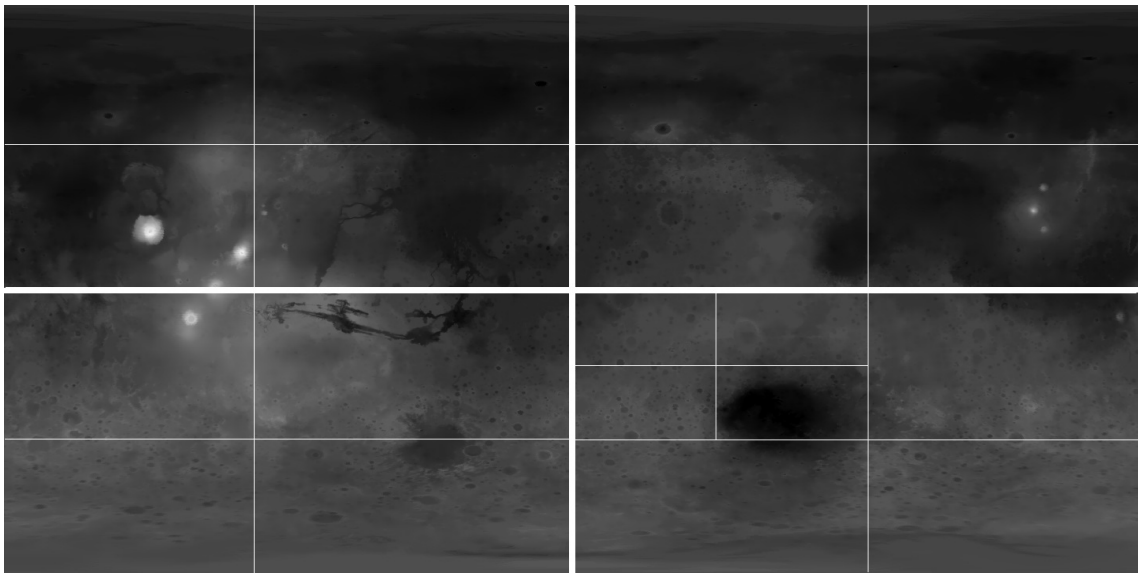

Figure 2.2: Subdivision of an image into a quadtree structure.

20]. This means that areas farther away from the viewer are rendered with lower detail whereas areas that are closer to the viewer are rendered with a higher detail. Therefore, processing time for farther away areas is reduced since the details of such areas are not discernible to the viewer. These algorithms are referred to as *adaptive*

because the LOD will change with the viewer's movements in the virtual world.

One of the main hardware advances in the past few years has been the programmability of the GPU, allowing access to the world of GPGPU programming [24]. This has allowed many algorithms to use the multi-core power of the GPU for parallel rendering of terrain. This leap forward has allowed many terrain renderers to render high-quality terrain in realtime by generating and maintaining a mesh that lies entirely on the GPU. For data caching purposes, the GPU can be used to speedup composition of multiple datasets into one image for use with a terrain renderer which generates and/or stores the terrain mesh on the GPU.

The rest of this chapter describes three papers that have addressed the problem of out-of-core data caching for terrain rendering. These papers were selected because they use many common out-of-core techniques and represent the body of the previously done work.

2.1 Multi-Resolution Deformation in Out-of-Core Terrain Rendering

2.1.1 Overview

Brandstetter *et al.* [5] present a data caching mechanism for swapping data both into core and out of core for deformable terrain. This is accomplished by a lengthy preprocessing step in which a dataset is split into a quadtree and stored on the hard drive for later rendering. As the quadtree is built, vertices are created from the heightmap data and stored in the child nodes of the tree. Each parent then represents a coarse version of its child nodes by removing data until the top of the tree is reached. Therefore, a simple LOD system can be used based on the distance of the viewer to a given patch of data to be rendered.

At runtime, a separate thread is launched to either load patches into memory or write them back out to the hard disk. This thread maintains a queue of terrain patches that are ready for rendering. Once a user-defined memory footprint has been

reached, the thread determines which nodes to write back out to the hard drive via a least recently used (LRU) algorithm. Therefore, terrain patches that have not been rendered for some time are assumed to no longer be needed and are discarded to disk.

As nodes are constantly being allocated and deallocated in system memory, a free list is also used to avoid the *new* and *delete* operators available to standard C++. This way, the time of waiting for the system to allocate a contiguous region of memory is removed. Additionally, this approach helps to remove the penalties of memory fragmentation that can arise from constant allocations and deallocations on the heap.

A resulting view of rendered terrain from [5] can be see in Figure 2.3.

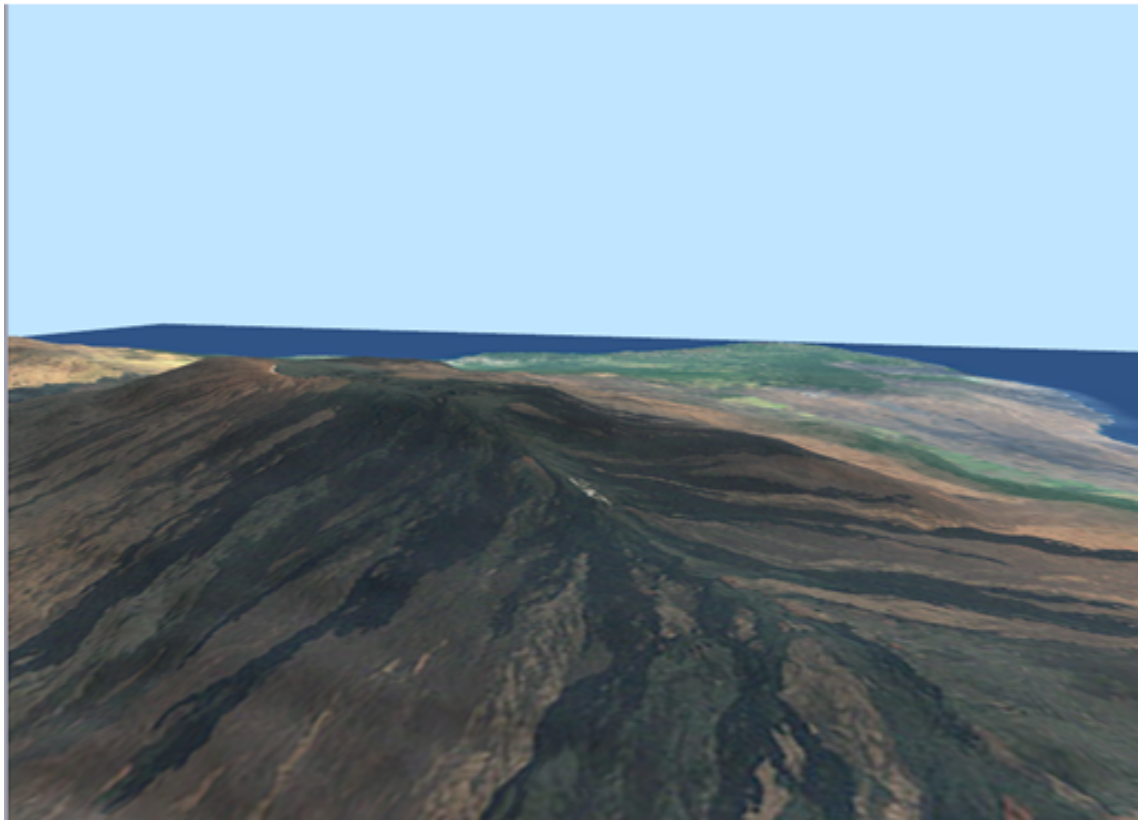

Figure 2.3: A rendering of Hawaii from [5].

2.1.2 Evaluation

The approach suggested is very useful for realistically modeling a certain area of terrain, allowing for realtime deformation, helping to alleviate the problems of dealing with large datasets that do not fit within system memory. Additionally, a simple LOD mechanism is used for rendering terrain patches at differing LODs. However, as this approach is only designed for one dataset at a time, it will not handle the problem of dealing with multiple datasets and compositing them over each other for a terrain renderer to use. Much of the algorithm is designed with planar terrain rendering in mind making the adaptation to a planetary body a very difficult process.

2.2 P-BDAM

2.2.1 Overview

Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM) [7], from 2003, is an extension to the BDAM [8] terrain renderer for planetary rendering of terrain. In this paper, the authors determine how to render terrain over a spherical body in realtime. An out-of-core data caching mechanism is used which takes the dataset for the planet and builds a texture hierarchy utilizing a quadtree data structure as well as the triangulated mesh during a preprocessing step. As is common, each level of the texture hierarchy has less detail than the level below it, constructing a mipmap pyramid for use in their LOD algorithm. Once preprocessed, the data is laid out in an optimal fashion on the hard drive to lower the probability of time costing page faults for faster rendering. Additionally, the optional DX1 compression scheme can be used to compress data on the hard drive so that less information needs to be read into memory from the disk. An LRU system is implemented to determine what patches of terrain can be safely discarded to the hard drive.

The LOD scheme used is based on the screen-space error of nodes in the geometrical hierarchy containing the triangle mesh. As the bottom of the tree contains the highest-resolution data and the parents contain coarser and coarser data, errors

accumulate from one level of the tree to the next. This error is calculated as the difference of the highest point of the terrain at one LOD and the highest point at the next LOD. Once projected into screen-space, the error can be determined. Utilizing an error threshold, nodes can be determined to be at an acceptable LOD based on their screen-space error.

During each frame render, the texture and geometry tree hierarchies are searched for data pertaining to the area around the viewer. Once data is found, it is uncompressed and uploaded to the GPU for rendering at the determined LOD. The GPU then combines the geometry and texture data for the final image, as shown in Figure 2.4.

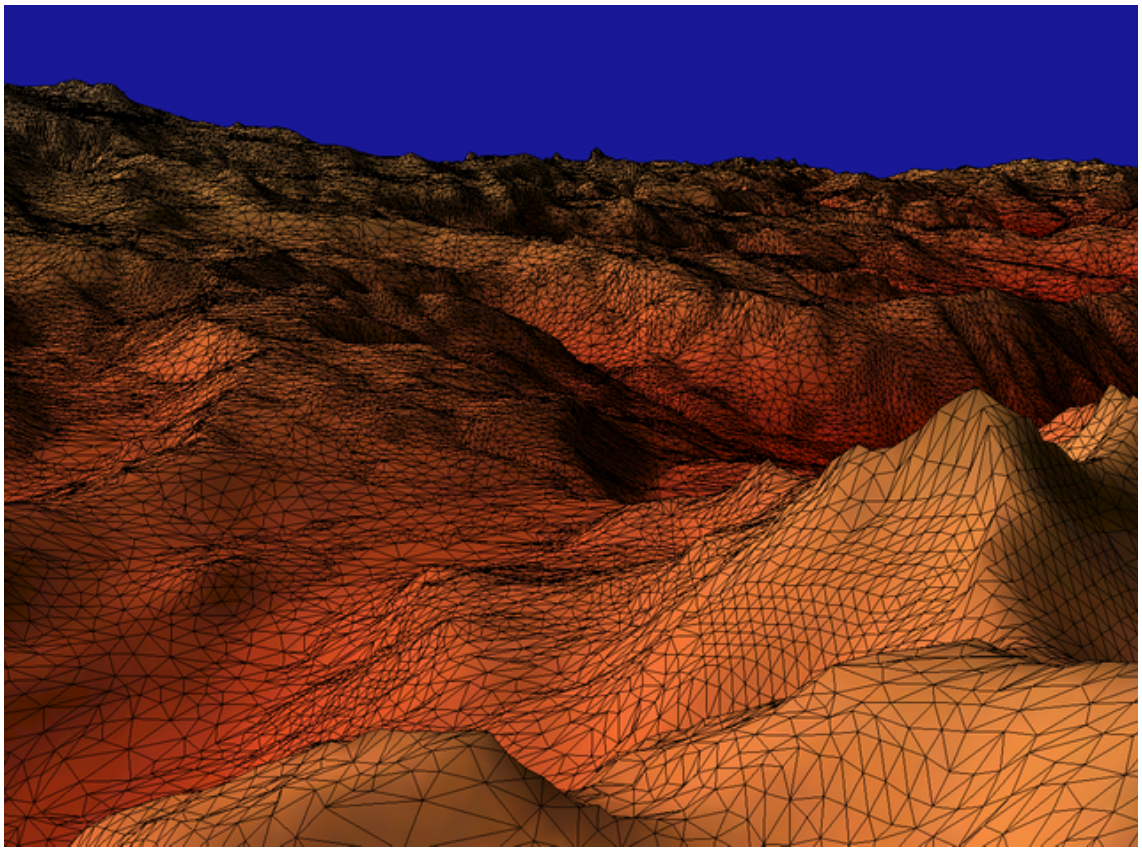

Figure 2.4: Resulting image from P-BDAM overlaid with the triangle mesh [4].

2.2.2 Evaluation

P-BDAM presents some interesting data-caching ideas. Compression of the terrain into smaller files which are laid out on the hard drive in an optimal read-in fashion can produce a great speedup for dataset chunk swapping. The LOD scheme is simple to implement with any dataset and quick to compute. However, as the algorithm only supports the use of one dataset at a time, the visual accuracy of the planet is limited as well as the usability of the algorithm as a planetary renderer. As their approach is triangle based, the addition of multiple datasets could be a challenge to implement because the combination of separate terrain meshes at runtime from one frame to the next could be difficult to implement, especially in realtime. Therefore, while this approach does deal with the data-caching issues of reading data efficiently from the hard drive, it does not address the problem of full planetary rendering for multiple datasets. Additionally, P-BDAM waits for all searches through its hierarchies to end before rendering which forces the terrain renderer to wait until all disk writes and reads have finished. Without the use of background threads to perform the data caching work, the terrain renderer's performance is limited by the hard disk speed.

2.3 Planetary-Scale Terrain Composition

2.3.1 Overview

Kooima *et al.* [15] present a full planetary terrain renderer with out-of-core support for both the terrain and textures. Each dataset is subdivided into a quadtree hierarchy and written to the hard drive for later paging into system memory. As each dataset has a rectangular shape, a rectangular bounding box can be created around them to serve as a container for any search mechanism. Using the viewing frustum of the camera, the datasets can be searched and paged into system memory. Once the pages have been loaded, they are drawn to a pixel buffer object (PBO) for later use on the GPU as a texture atlas. These textures are then composited over each other and used for geometry mesh generation directly on the GPU, the result of which can be seen in

Figure 2.5. Along with the texture atlas, a different texture is created which details the locations of individual terrain patches in the texture atlas. Therefore, the GPU has knowledge of where each terrain patch exists in the texture atlas for composition into the final terrain image. As is common for out-of-core data-caching algorithms, an LRU approach is used to determine which data patches should be left in memory and which should be discarded back to the hard disk once a predefined memory threshold has been hit.

A simple screen-space error metric is used to determine the acceptable LOD of a given terrain patch, as described in [7]. As the viewer moves closer to a dataset, it is rendered with a higher LOD. This process is performed so that compute resources are not wasted in rendering details of far-off datasets that cannot be seen by the viewer.

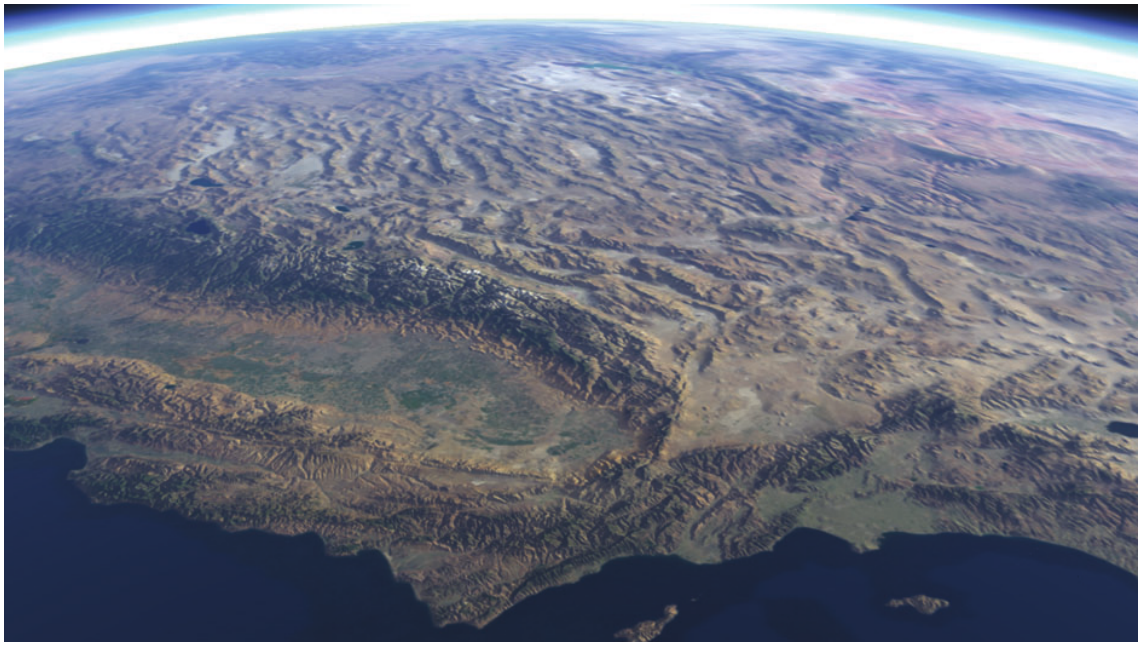

Figure 2.5: Resulting image from [15].

2.3.2 Evaluation

The planetary terrain rendering algorithm proposed by Kooima seems to accurately deal with the nuances of rendering an entire planet. The creation of a texture atlas

serves to simplify the combination of multiple datasets into one final image which the terrain renderer can then use for mesh generation. Additionally, the common use of the quadtree as the hierarchy for splitting datasets into chunks is performed as in many other out-of-core rendering techniques, simplifying the processing of regions of the dataset as well as providing a simple mechanism for displaying multiple LODs. It would seem that all of the problems of dealing with planetary terrain rendering have been solved here. However, as it is unspecified as to how the multiple datasets are ordered and searched through (aside from knowing that the datasets form a box) we cannot make any assumptions as to how the data is laid out on the hard drive and therefore how a terrain renderer can interface with the data-caching mechanism. Additionally, each frame waits for all datasets to be searched through, limiting the speed of the terrain renderer to that of the searching process. Lastly, this algorithm does not support the addition of new datasets at runtime which would force the user to recreate the entire texture cache hierarchy in order to display new data, which could be an extremely lengthy process depending on how much data would need to be reprocessed.

2.4 Summary of Previous Work

We have presented an overview of several previous works in the area of out-of-core data caching for planetary terrain rendering and have evaluated them based on the problems that they both do and do not solve. Each algorithm presented [5, 7, 15] deals with the problem of datasets that are too large to fit in system memory and therefore must be partitioned and stored on the hard drive. As shown above, the most common way of doing this is using a quadtree hierarchy and creating a mipmap pyramid which can also serve as a simple mechanism for storing data at different LODs. In terms of multiple datasets, the only one that solves this problem is [15], which also proposes a simple way of compositing them together on the GPU. Each of these works is missing some important aspects of planetary terrain rendering such as ordering multiple datasets on the hard drive and efficiently searching through

them, use of CPU parallel processing to decouple the renderer from the data caching mechanism, the ability to add new datasets to the data cache at runtime, and a generalization to work with either a triangle or ray-based terrain rendering approach. In Chapter 3, we will present our ideas for handling such problems.

Chapter 3

Out-of-Core Data Management for Planetary Terrain

3.1 Overview

In this chapter we present an out-of-core and level-of-detail algorithm that can be integrated into a planetary terrain renderer. We combine the approaches described in Chapter 2 in order to accomplish this by dealing with both datasets that are too large for system memory and a hard drive limited number of datasets. As well, we will extend this approach in order to decouple the data-caching mechanism from the renderer, allow for new data to be added at runtime, and overlay multiple datasets on top of each other.

As the number of datasets is suitably large, we will need to have a non-trivial preprocessing step which creates the hierarchies for each dataset and orders them into a BVH. After the data has been preprocessed, it is ready for use by a terrain rendering algorithm. Skeleton versions of the hierarchies, which have no image data, are loaded into main memory in order to perform searching operations. Once data has been found, it is loaded into a queue of terrain patches which are then uploaded to the GPU for composition. We sort the terrain patches based on their distance to the user. When the user-defined memory threshold is hit, patches are removed from the queue as to not exceed the system resources. As our algorithm takes advantage of parallel processing on the CPU, all of the above operations can happen simultaneously, making the rendering algorithm independent on the performance of the searching

algorithm and the memory hardware. A graphical overview of the algorithm is shown in Figure 3.1.

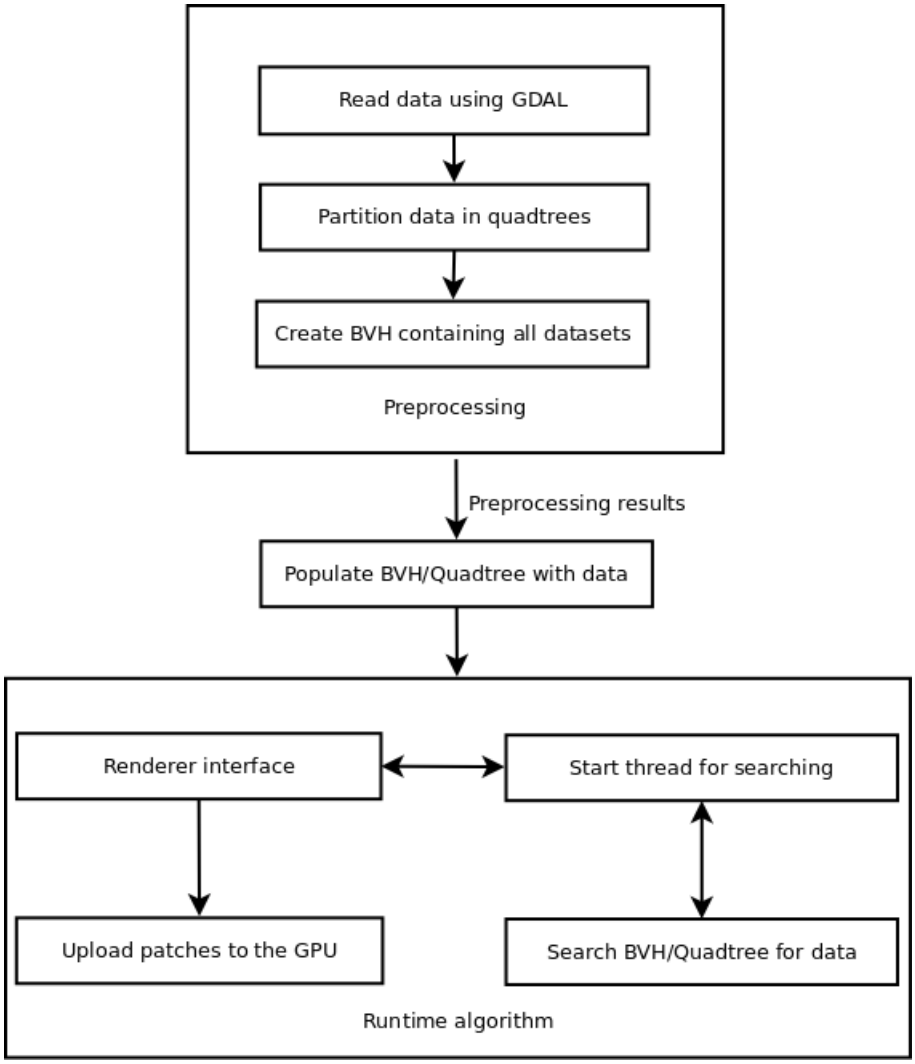

Figure 3.1: Flow chart depicting the various stages of the proposed algorithm.

3.2 Preprocessing

As the number of datasets is suitably large, a non-trivial preprocessing step must occur in order to get the data ready for both searching and rendering. Since the data does not change once this process has happened, preprocessing only needs to be run once and not each time the algorithm is used. Therefore, the datafiles output by the

preprocessing step can be copied to any runtime version of the algorithm and used for rendering. The preprocessing step contains three separate phases:

1. Break all datasets into smaller chunks based on a user-defined maximum image size using GDAL (Section 3.2.1).

2. Once the datasets are broken down, they are placed into a quadtree hierarchy. This allows for the creation of a mipmapping pyramid of data for the LOD scheme to use based on the resolution of the data for a given dataset (Section 3.2.2).

3. Order all datasets into a BVH for searching purposes. Once the structure has been created, it can be written to the hard drive (Section 3.2.3).

3.2.1 Geospatial Data Abstraction Library (GDAL)

GDAL [14] is an open-source library which can be used to extract regions of an image along with the geographical data pertaining to that specific sub-region of the dataset. We use this libray to get dataset specific information such as the local radius of the planet, geographic coordinates of each corner of the dataset, and the central latitude and longitude for projection purposes. Additionally, this library is used to extract pixel data for each leaf node of the quadtree hierarchy.

Most datasets will also contain pixels which are actually not part of the dataset but are included to make the dataset appear as a rectangular image. These pixels will contain what are known as *no-data values*. Each dataset can potentially use different numbers to represent its no-data value so we extract this information from GDAL. Any pixels being processed that equal this value are discarded from further processing so that they do not distort the final image.

3.2.2 Quadtree Creation

A quadtree is a spatial subdivision hierarchy which is defined as all nodes either having zero or four children. If a node needs to be subdivided, it is done so into

four equal-sized children. Figure 3.2 shows a hierarchical view of a quadtree as well as the subdivision of space that is represents. Any nodes that do not have children are declared to be leaf nodes. This hierarchy was chosen because it allows for both a simple mipmapping algorithm and LOD selection for the data contained within the tree. Each node contains data either at the resolution of the original dataset or at the resolution of an average of its children [15].

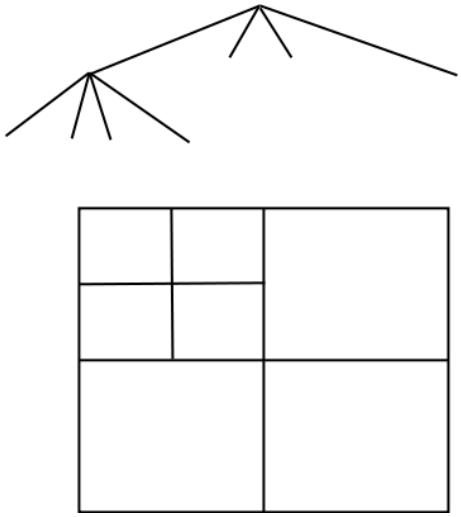

Figure 3.2: Quadtree views.

As each node of the tree is created, we create a bounding sphere around it that is centered at the center of the node with a radius which is as long as half of the distance of the longest axis of the dataset. Using this sphere, we can perform simple sphere-frustum culling to determine if a given patch of data is in view. We use the sphere because, without any knowledge of the terrain that a data patch represents, it can be used to contain any geometry which might protrude along the dataset's normal. Therefore, a high-quality mesh is visible for regions of data where the tip of a mountain is directly in front of the user but the base of the dataset which contains the mountain is not (Figure 3.3). Additionally, we represent the sphere in world-space coordinates so that no special operations need to be applied to the camera frustum in order to intersect with the tree.

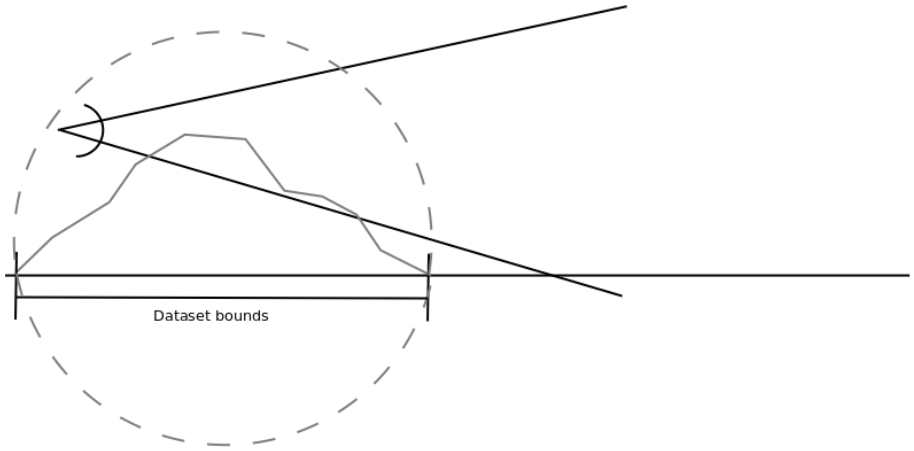

Figure 3.3: Terrain features such as mountain tips can still load high-quality data.

In order to speedup the later composition step, we can create the transformation variables from projection to texture coordinates now as opposed to having to recreate it for each composition pass. To do this, we can use the geotransform matrix provided by GDAL per dataset. This 2x3 matrix represents the transformation of pixel coordinates into projection coordinates for a given dataset. Using this, we can calculate L and U, the lower-left and upper-right corners of the dataset respectively, by converting their pixel locations. Once we have these, we can determine the width and height of the dataset S in projection coordinates and store these for use by the compositor by subtracting U from L. During composition, these values are used to calculate texture coordinates as shown in Figure 3.4. These texture coordinates for the given point G are calculated by determining the relative position D of G to L. For each component of D, we can determine the texture coordinates (s, t) by D/S.

These values can be calculated and stored per data patch for reading later. In order to increase GPU performance, we store the inverse of the dataset height and width in projection coordinates therefore preventing the GPU from performing costly divides for each fragment of the dataset.

Our quadtree is created with a bottom-up approach, meaning that the leaves of the tree are created first, followed by the parents. This is done so that the highest-resolution data is stored in the leaves of the tree while the parents store lower and lower

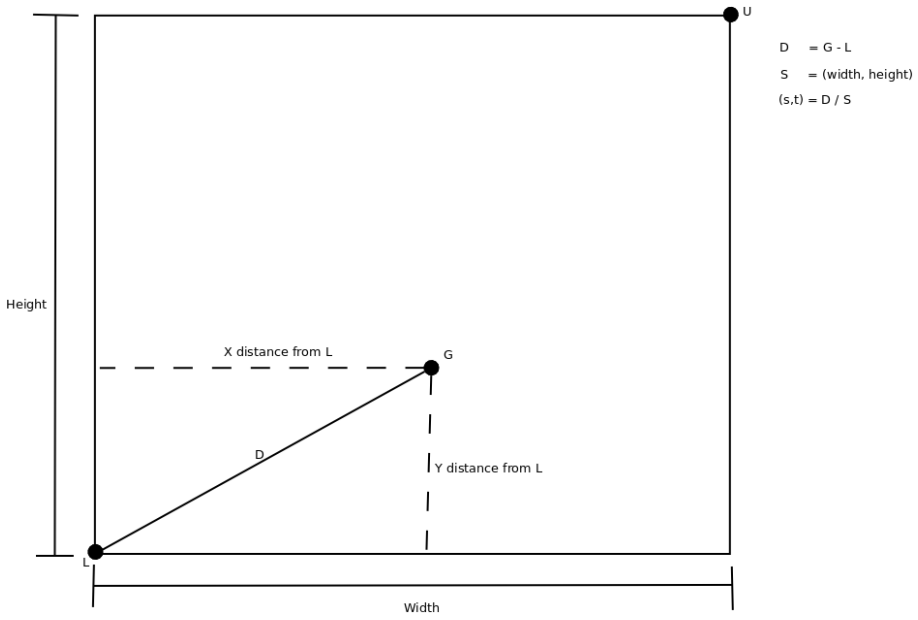

Figure 3.4: Construction of texture coordinates for the point G.

resolution imagery. This type of hierarchy is known as a mipmap [1]. A mipmap is defined as each node having either the highest-possible resolution data or an average of its children. For the parent node, each pixel of image data is the result of the average of the four corresponding pixels from its child nodes, resulting in a lower-quality image that still covers the same area (Figure 3.5). As shown in Figure 3.5, the four child nodes contain high-resolution imagery of a region of a heightmap. The parent node then contains the same regional area but at a lower resolution.

Once the build reaches the root node, it creates the lowest-resolution version of the data and finishes. As nodes are created, they are written to the hard drive to not overload the system memory for datasets that are too large. Along with the node data itself, a table file is written to the hard drive that details the structure of the particular dataset. This includes data such as the bounding sphere for the whole dataset, projection information, and offsets into the node file detailing exactly where the image data for a particular node is located. This file can then be used at runtime for quick access to per-dataset information.

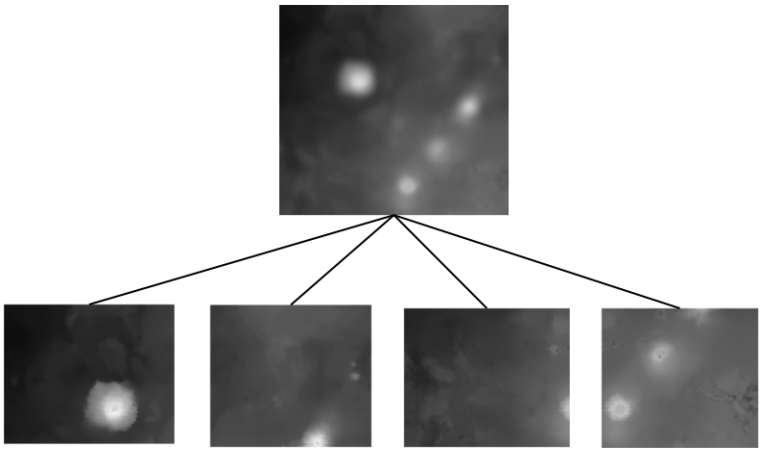

Figure 3.5: Mipmap hierarchy.

3.2.3 BVH Creation

After all datasets have been preprocessed, we then turn our attention to building the BVH which contains all of the datasets so that they can be efficiently searched. The BVH is constructed as three-dimensional axis-aligned bounding-boxes (AABBs) which are partitioned based on the spherical datasets that they contain in world coordinates (Figure 3.6). Each node determines if it is either a) done splitting and can stop or b) needs to continue splitting the data further so that less datasets are searched through per node. If a node decides that it needs to split, it creates two equal sized bounding boxes which together equal the volume of the parent and inserts the parent's data into the respective child. Once all data has been moved, the child nodes then shrink themselves to tightly contain the datasets within them, effectively culling empty space in the tree from being searched. As each node is created, it serializes itself to the hard drive for later reading at runtime. The preprocessing step is over when the BVH creation process has finished.

3.3 Runtime

In order to utilize the cached datasets at runtime, the first thing that needs to happen is to read the data files for the structures of the datasets on the disk. A skeleton version

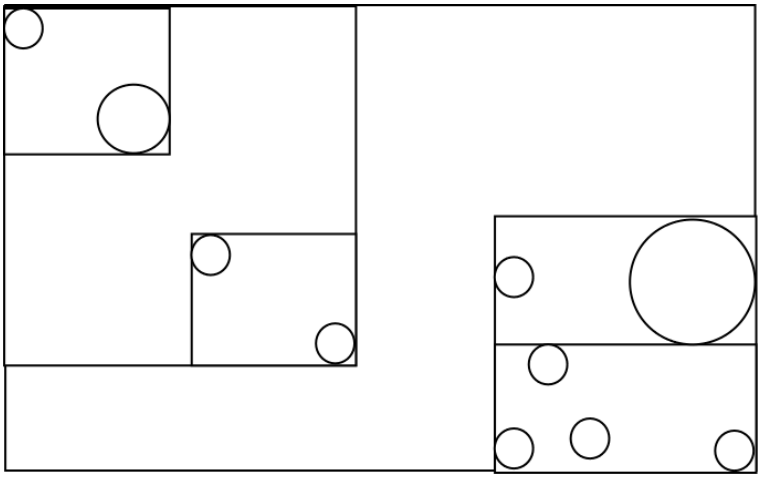

Figure 3.6: BVH containing spherical datasets, represented in two dimensions.

of the BVH along with each dataset hierarchy can be read into memory and used for searching. This means that the structure of the trees is loaded but, until needed, the data for each patch of terrain remains on the hard drive. In order to make sure that the reading of these hierarchies does not interfere with the initialization of the terrain renderer, this process is performed on a separate thread. At any time during this phase, the renderer will be able to make queries to the interface to determine the status of this operation. Once all necessary data has been read in, we are ready to move into the main phase of the runtime algorithm, which contains five separate steps:

1. Search the BVH and quadtree hierarchies to determine what is visible (Section 3.3.1).

2. Determine the appropriate LOD for a dataset currently being searched (Section 3.3.2).

3. Upload the data to the GPU and composite all visible datasets into one image for use by the renderer (Section 3.3.3).

4. Allow for the insertion of new data (Section 3.3.4).

5. Maintain a list of data patches that are currently in system memory (Section 3.3.5).

3.3.1 Search

In order to perform a search, the renderer must provide the current camera's viewing frustum. This frustum is copied into the search object and sent to the BVH for searching. As the BVH is constructed from AABBs, a simple AABB-frustum collision detection algorithm can be used to traverse deeper into the tree. Once the search comes across a node that contains datasets, each dataset is searched by using the camera's frustum if and only if two conditions are met:

1. The frustum intersects with the dataset's bounding sphere.

2. The dot product of the dataset's normal and the inverse viewing direction is greater than zero. This test is performed to exclude datasets that intersect the viewing frustum but are on the other side of the planet relative to the viewer. If, however, the dataset is a global dataset containing data for the whole planet, this step is skipped.

Each dataset can be tested simply using a sphere-frustum collision detection algorithm. As a parent is intersected, its children are then tested until an acceptable LOD has been determined (Section 3.3.2). Once the proper LOD has been found, that patch of data is loaded from the hard drive and saved in a list of newly found patches. If the search algorithm comes across a node that is already loaded into gpu memory, it will not reload it.

As the search operation can become costly (especially if a majority of found patches need to be loaded from the hard drive), it is entirely run by a background thread. Utilizing this approach, we are able to reduce the dependency between the terrain renderer and the data cacher. Until new data is available, the renderer can simply render the terrain with the data already cached from the previous frame.

3.3.2 LOD Selection

We adapt the approach in [16] to determine if a level of the quadtree hierarchy has an acceptable LOD for the current position of the camera. As this approach is for geometry, a simple change to read image data was necessary for adaptation. This method utilizes screen-space errors to determine if an image is of valid quality or not for the user based on projecting each data patch into screen space and comparing it with a user-defined maximum error. As the mipmapping process during preprocessing happens (Section 3.2.2), errors will accumulate from the averaging of data. This error, represented as δ_m, can be calculated by the difference between the highest point in the child data and the highest point in the new averaged data. As suggested in [16], we need to ensure that as the data gets coarser, the error gets higher. To do this, we can add the child error the current parent node. Therefore, the actual error for a node δ_c can be determined as given in the following equation.

$$\delta_c = \begin{cases} 0 & \text{if leaf node} \\ max(\delta_{c0}, \delta_{c1}, \delta_{c2}, \delta_{c3}) + \delta_m & \text{otherwise} \end{cases} \tag{3.1}$$

As the value calculated in Equation 3.1 does not change, we can safely perform this in the preprocessing step and serialize it to the hard drive. However, at runtime we need to calculate the final error for each node. For this, we need the height of the screen in pixels S, and the field of view of the camera. These parameters can be supplied to the tree builder at runtime so that the errors can be calculated.

In order to perform the projection into screen-space, the following equation will be used as presented in [16].

$$\epsilon = \delta_c \frac{S}{2\delta_m |\tan \frac{fov}{2}|}$$

Using this equation, we can determine the minimum distance d_m to the terrain patch given the user-defined error threshold τ. We can then compare d_m to d (the distance to the camera from the terrain patch) while searching. If d is less than d_m, a

higher LOD is needed because the error at this patch is too large. Therefore the child nodes will be tested until the end of the tree is found. We calculate d_m per patch by determining the distance d where ϵ is equal to τ by substituting τ for ϵ. Therefore, we end up with a final equation for δ_m which we can calculate per node and store for later comparison.

$$d_m = \delta_c \frac{S}{2\tau |\tan \frac{fov}{2}|}$$

If, at the bottom of the tree, the error is still too large, the child nodes will be added into the queue of terrain patches for rendering as we cannot supply any higher-resolution data.

3.3.3 GPU

In order to speedup the composition of multiple datasets, we rely on the multi-processing power of the GPU. As new data becomes available from searching through the dataset hierarchies (Section 3.3.1) we need to composite them together so that the terrain renderer can use a singular image to generate a three-dimensional mesh. To do this, we need to create what is known as a *texture atlas* [15] which is all of the currently loaded data stored in one place for easy access by the GPU. An example of a texture atlas for color data can be seen in Figure 3.7. In order to create the atlas, we start with a blank texture and upload each dataset to it individually. Thus, each dataset covers a region of the texture. As this data is stored in a texture, the underlying hardware specifies a rigid upper-bound for the size of this atlas, which is the maximum user-defined memory value that can be used. Modern graphics cards offer a maximum texture size of 8192 pixels per dimension [25]. Using the OpenGL API we are able to get this value at runtime so any future cards which allow larger texture sizes can be utilized without any need to change the algorithm.

While uploading the data to the atlas, we also create a buffer which is sent along with the atlas texture to the GPU. This buffer contains information pertaining

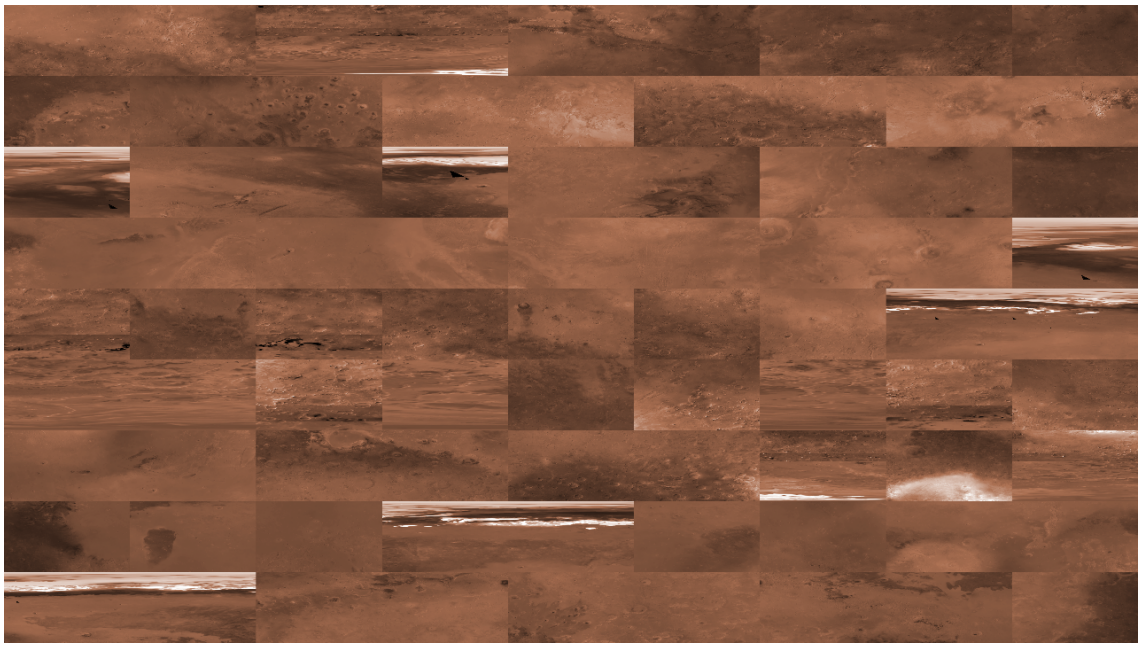

Figure 3.7: Texture atlas of loaded color datasets.

to the individual datasets for use by the GPU composition algorithm. This data details the (x, y) location of the start of a dataset in the atlas and how large it is, the center latitude, longitude and local radius for the dataset, what type of projection the dataset is stored in, and finally the texture coordinate transform variables described in Section 3.2.2.

Once the data has been uploaded, the GPU takes over for the composition phase. Using programmable shaders, we can read the texture atlas and composite the data into a new texture. This is done by rendering, for each dataset in the texture atlas, a screen-aligned quad which tightly fits the data. This new quad along with its respective region of the atlas is sent to the fragment shader and drawn into the output texture. If any regions of the data overlap, their data can be blended to create the final texture. Once all datasets have been added to the final texture, the GPU can then be used for mesh creation by the terrain renderer.

3.3.4 Insertion of New Data

It may be possible that at runtime, a user will want to see different data that has not already been cached on the hard drive. Our algorithm supports this process and is one of the main reasons we chose to use a BVH as our main spatial subdivision structure as the addition of new data will create nodes that can simply be refit into the existing hierarchy without having to recreate the entire tree.

To add new data, the data cacher will spawn a new thread to process it. This way, the system rendering and data searching is not interrupted by the processing of the new data. In order to get the new data ready for rendering, it must be processed as in Section 3.2. Once the quadtree hierarchy has been created, the dataset will need to be fit into the existing BVH. This is done by determining what node the new dataset fits into and inserting it there. If the new node has been overloaded by the addition of the new data, it is split into two separate nodes and the data contained within is stored in the newly created children. As the algorithm cannot make assumptions about the intentions of the user, the data cacher is able to be passed a flag which determines whether this new data is serialized to the hard drive or not. Therefore, the newly processed data can be added to the existing data cache or it can be discarded upon program exit.

3.3.5 Maintenance

As data is being added to the list of renderable patches, it can grow too large for the GPU memory to handle. Additionally, there can be patches of data still loaded into memory that are on the other side of the planet and not relevant for the current frame. Therefore, we need a maintenance step to be performed in order to discard old data back to the hard drive for future use.

To implement this functionality, each patch of render data needs to maintain a value which denotes how far it is from the user. On each update step, the queue is reordered based on the patch's respective distance, with the closest patch at the head of the queue. Once the user-defined maximum amount of memory has been

taken up by the patches loaded, we will need to remove patches from the queue until the memory taken by the queue no longer exceeds the maximum. This can easily be performed by removing the end element of the queue and checking the memory usage of the new queue against the maximum.

As the reordering and removal of data from the queue of patches to render can potentially be slow, we implement this process on a separate thread so as to not disrupt the renderer from generating terrain.

3.4 Context-Safe Rendering

In order to interface with a renderer which works in a virtual reality (VR) environment, we need to be able to upload data to different textures to be used as a respective texture atlas per eye. This is accomplished by the user of a context identifier being passed into the GPU upload routine so that the data can be placed in the proper texture. As all of the search operations are performed by background threads, multiple searches can happen simultaneously meaning that both searches for the two eyes can happen at the same time and not ruin performance. Aside from the need to deal with the GPU per render context, no changes need to be made to the algorithm for the adaptation into a VR environment.

3.5 Global Height Dataset

Should the user wish, they can have the system load any global height dataset that they wish. This dataset resides in system memory and is loaded at runtime using the same code as the rest of the system. It is never written to the hard drive nor does it have any mipmapping process performed on it. Using this dataset, the user can query the height of the terrain at any given latitude and logitude coordinate. This can be used for rendering operations that do not rely on specific dataset values but instead a global approximation of the planet. Since it is up to the user to tell the data cacher what dataset to load in, the resolution of the image is dependent on what

they require. Currently, this is only available for height data.

Chapter 4

Implementation

This chapter describes the implementation of our algorithm as a data-caching library for use with the terrain renderer Hesperian [21]. We describe both the functional and non-functional requirements of the implementation, use-cases, and a description of the classes which makeup the library.

4.1 Requirements

Our algorithm is designed to work across different display types as well as different operating systems. In order to do this, we must make our implementation both thread-safe (via locks [18]) and context-safe (Section 3.4). Additionally, we make use of cross-platform APIs [9] such as OpenGL [2] and GDAL (Section 3.2.1). This way, our library can be used on any operating system/display environment without any change to the algorithm, making it more applicable to any terrain renderer.

For our implementation we chose to render the planet Mars. However, because we use GDAL to determine the projection information for all of our datasets, any spherical body can be rendered if the data is available. Most of this data, which is represented as either height or color data, can be obtained directly from NASA. If the projection information exists however, any data can be used. Additionally, as NASA has many missions to map similar regions of planets, there is a large possibility of there being overlapping data for a given region. For this reason, we have chosen to implement a composition algorithm in order to accurately deal with overlapping

datasets.

At any time, the user is able to add new data to the data-cache for viewing. Once processed, this data will be shown to the screen if the viewer is in the area where the new terrain is located. As the data is searched, the proper LOD (Section 3.3.2) is determined and used, allowing for a continually adaptive LOD rendering of the terrain. Since we do not want the data-cacher to slow down the user-experience of realistically rendered terrain, we utilize the multi-core power of modern CPUs and push all searching and uploading of patch data to separate threads.

Using common projection equations, as given by Eliason [13], we are able to place any dataset in the proper geographic location. We support both equirectangular and polar stereographic projections. As different data can be stored in these projections, it is imperative to use the proper projection when transforming the coordinates into texture coordinates or the image will have continuity problems.

These functional [29] and non-functional [26] requirements are listed in Table 4.1 and Table 4.2 respectively.

Table 4.1: Functional requirements.

F01	The library will read a standard data format for height, color, and normal data.
F02	The library will allow the data patch size to be changed for different hardware.
F03	The library will allow the addition of new data at runtime.
F04	The library will allow for new data to be added to the data cache.
F05	The library will composite overlapping datasets.
F06	The library will accept a maximum screen error for LOD selection.
F07	The library will accept a maximum amount of system memory to take up.
F08	The library will allow the user to preprocess terrain data.
F09	The library will select the proper LOD at runtime.
F10	The library will display datasets in the correct geographic area.
F11	The library will use threads to decouple the data-cacher from the renderer.
F12	The library will allow for out-of-core terrain rendering.

Table 4.2: Non-functional requirements.

N01	The data-cacher will be implemented as a library.
N06	The library will be implemented using C++.
N02	The library will be thread safe.
N03	The library will be rendering context safe.
N04	The library will use OpenGL.
N05	The library will use GLSL.
N06	The library will use GDAL for loading height and texture data.

4.2 Use Cases

The user of this application is considered to be a terrain renderer. Therefore, we provide a simple interface into our data-caching mechanism library for easy use of it's complex features.

Prior to any terrain renderer being able to use the data cacher, a preprocessing step must be performed which places the data into a state which is suitable for rendering. This includes building mipmap hierarchies, the dataset BVH, and pre-determining dataset errors (Section 3.2). This functionality is simple to use in our library as the interface can accept a list of datasets to preprocess. Once this step has completed, the renderer is able to use the data cache for rendering purposes.

At runtime, the terrain renderer can add new data to the data cache which can either be written to the hard drive or discarded upon program exit. Additionally, the application can search for new data to render and have that data composited into one final image for use in mesh generation. At program initialization, the renderer can specify the maximum amount of memory to be in use by the data-caching system as well as the maximum screen-space error allowable for LOD selection.

These use cases [28] can be see in Figure 4.1.

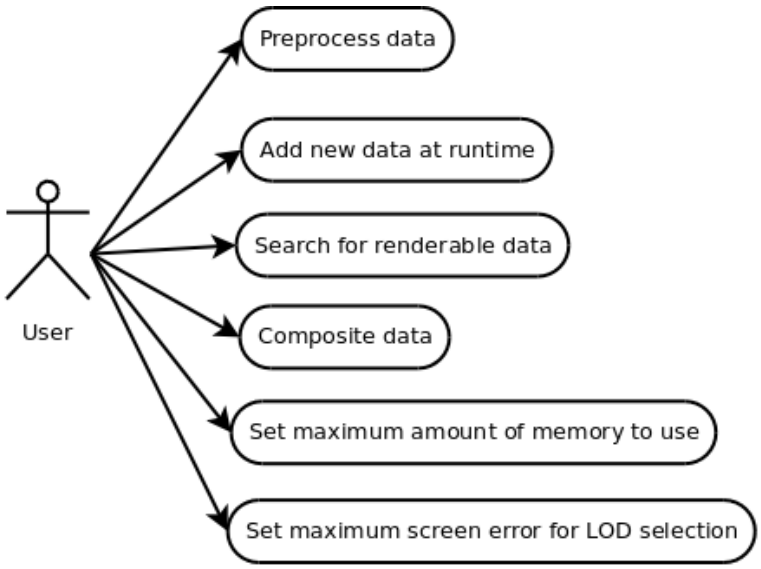

Figure 4.1: Use cases for the data-caching library.

4.3 Classes

As we have strived to maintain usability and simplicity in the use of our implementation, we have written the code in as few of classes as possible. For easy integration, the user only interacts with one class which is used to hide the complexities of the classes beneath it. An overview of the implemented classes can be seen in the class diagram [31] in Figure 4.2.

The `DataCacher` class acts as an interface into the rest of the system. With the class, the user can preprocess data, search for renderable terrain patches, insert new data, and set the required parameters for library use. At runtime, we initialize our data by reading the input files and building skeleton versions of the data structures without any loaded image data. Both the structure of the BVH as well as the datasets are stored in a file for reading. As each node is built into the structure, the accompanying datasets are read and created in memory. Each dataset contains a `TextureQuadtree` which contains information about the various levels-of-detail that each dataset supports as well as projection data specific to the dataset. Utilizing the `ProjectionParser` class, we are able to obtain this projection information with ease.

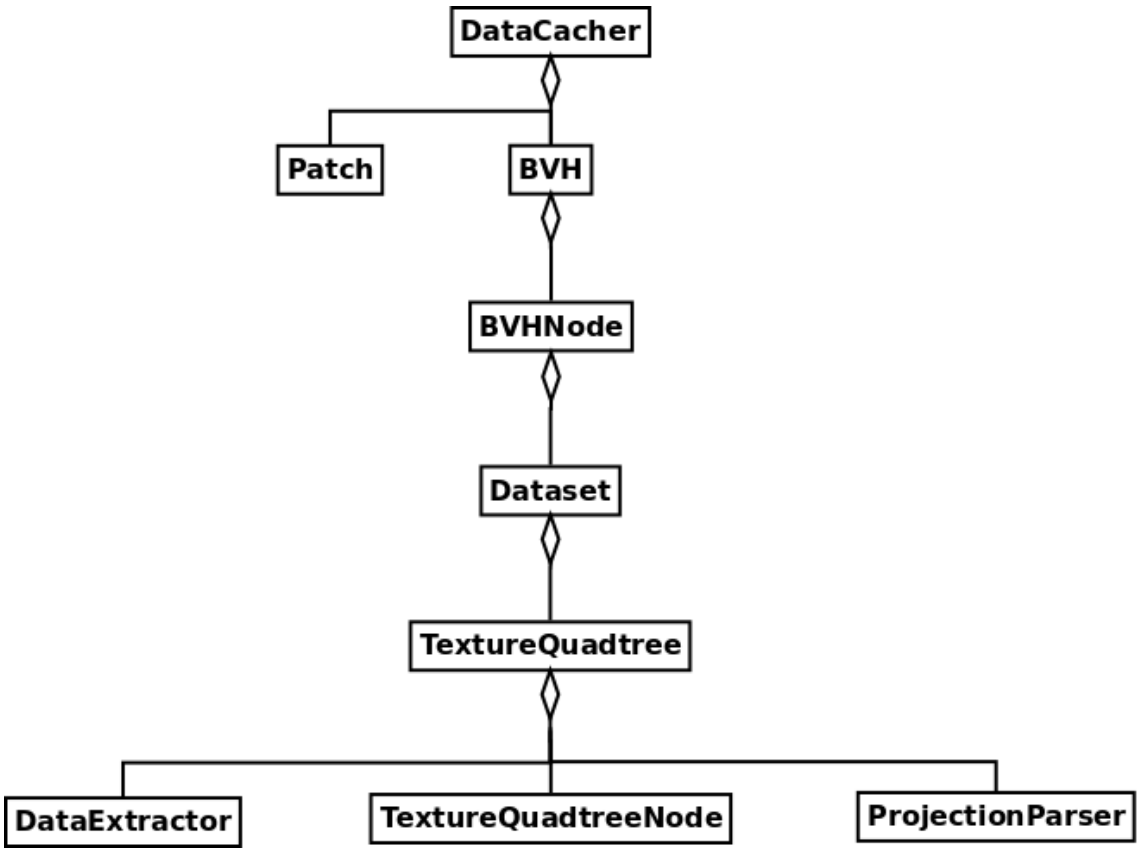

Figure 4.2: Implemented class diagram for the data-caching library.

To simplify and modularize the code, all GDAL commands are contained within the `DataExtractor` class. Therefore, when anything needs to be ascertained from the original dataset file, this class can oblige. For instance, this class is capable of providing the projection information, regions of pixel data, and the coordinates of the corners of the dataset in either projection, geographic, or world coordinates. As the original dataset is used as input for this class, it is only used during the preprocessing step and ignored during runtime.

In order to support rendering-context safety, we allow for multiple patch queues to exist (one per context). This way, different displays that are looking in different directions in the virtual world are able to load in their own relative data. To make this process work, we also need to implement the search algorithm using different search threads per context; therefore, each context can be searched simultaneously

without any holdup by one display in the system. We support this type of runtime environment by utilizing per context data (Section 3.4) which segregates the data for each context into separate instances of any class that needs to be context-safe. As an example of the type of display that would need per context data, we can look to the Cave Automatic Virtual Environment (CAVE) [30], an example of which is shown in Figure 4.3.

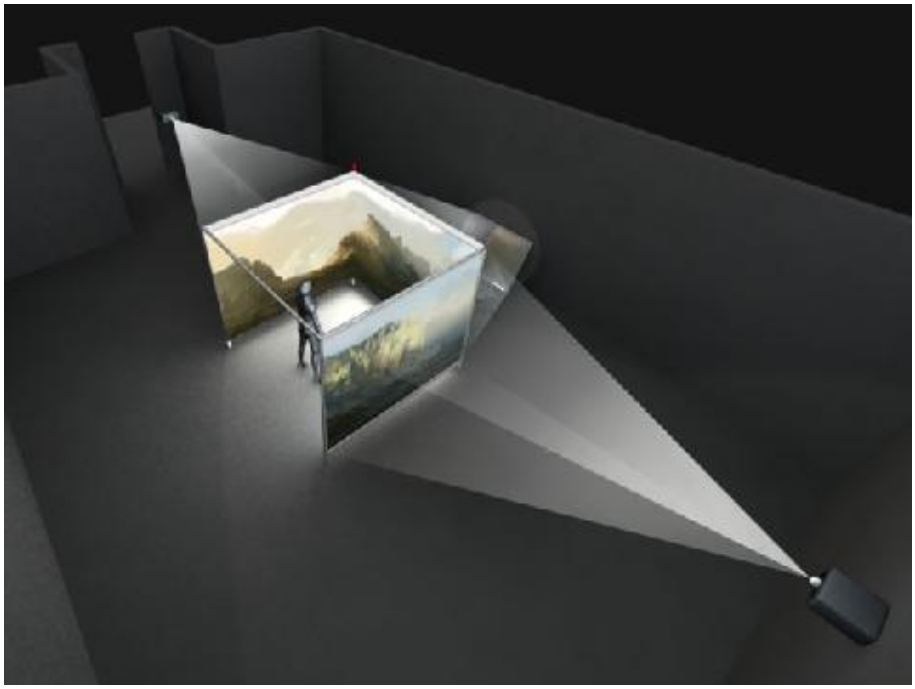

Figure 4.3: A CAVE display environment which utilizes multiple rendering contexts for visualization [6].

Outside of providing just an interface, the `DataCacher` class also takes care of maintaining the patch queue by determining if too much memory has been consumed. Old data patches are discarded to the hard drive until the system is no longer using more than the maximum amount of memory should this event occur. For rendering purposes, this class also uploads all patches to the GPU by the creation of a texture atlas. While the atlas is being uploaded, a legend is also being created which details where each dataset resides in the atlas. As a final step, the center of each dataset in world coordinates is saved in a vertex buffer object (VBO) for rendering. Once the

data for the atlas has been uploaded, the GPU takes over for the final steps of the implementation.

4.4 GPU

Using programmable shaders [27], we are able to speed up the composition step by use of the GPU. OpenGL gives us easy access to the geometry and fragment shaders which we can use for texture overlay options in order to create a resulting texture that the terrain renderer can use for mesh generation. As mentioned in the previous section, a texture atlas along with a legend for the atlas and a VBO of points is created for use by the GPU. Both of the shaders work together to produce the final image.

4.4.1 Geometry Shader

The geometry shader in the graphics pipeline can be used to turn incoming points into solid geometry. For our purposes, we transform the points defining the center of each dataset into screen-aligned quads which are received by the fragment shader. An activity diagram [3] showing the flow of events in this algorithm is shown in Figure 4.4.

Once the quad has been generated, it is automatically sent to the fragment shader by the GPU.

4.4.2 Fragment Shader

Using the fragment shader, we can directly affect the outcome of a pixel color on the currently-bound frame buffer. Each fragment of the quad output from Section 4.4.1 gets processed as shown in the algorithm detailed in Figure 4.5.

Using the world-space normal of the sphere at the current fragment, we can determine the geographic coordinates of this pixel. These coordinates can then be turned into projection coordinates based on the type of projection this dataset was created in. Using the newly calculated projection coordinates, we can generate texture coordinates as shown in Section 3.2.2. Once we have obtained the texture coordinates,

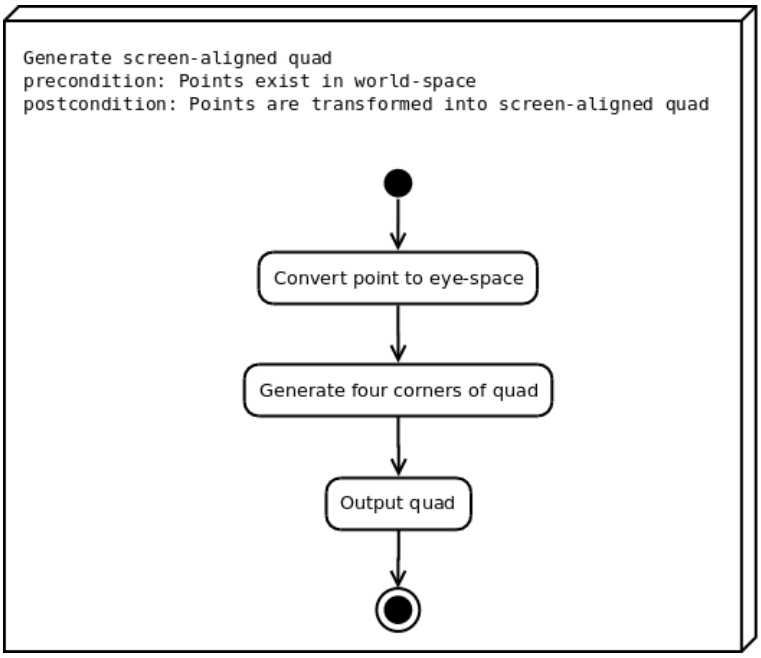

Figure 4.4: Activity diagram showing the generation of screen-aligned quads per dataset.

they need to be translated into atlas-specific texture coordinates, which can be done with the following equations for s and t respectively.

$$s = \frac{(T_s * D_{width}) + D_x}{atlas_{width}}$$

$$t = \frac{(T_t * D_{height}) + D_y}{atlas_{height}}$$

where T is the texture coordinates calculated specific to the dataset and D is the legend element for this dataset which contains the (x, y) position of the dataset in the texture atlas as well as the width and height.

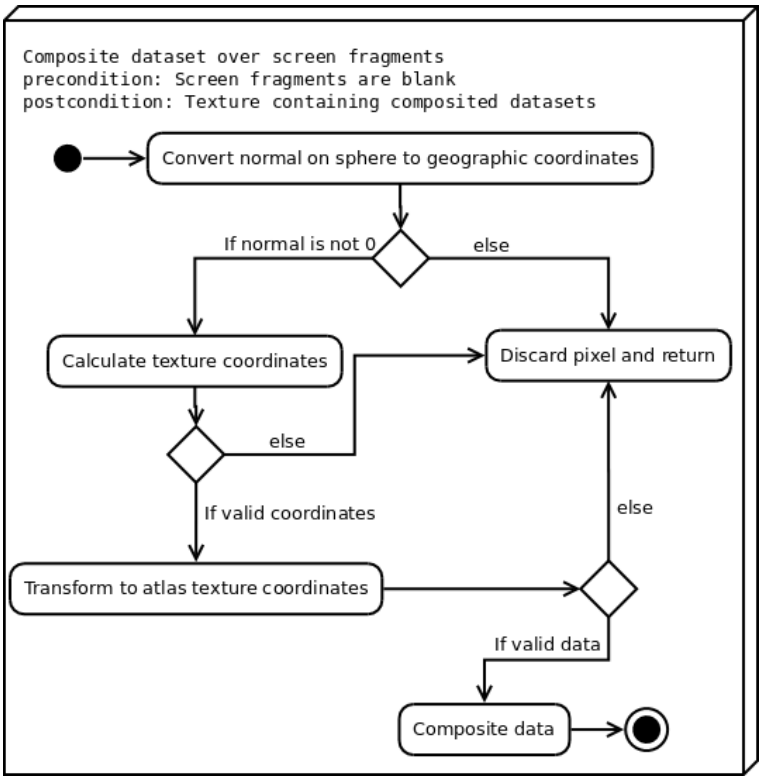

Figure 4.5: Activity diagram showing the algorithm used to composite datasets into the final texture.

4.5 Deployment

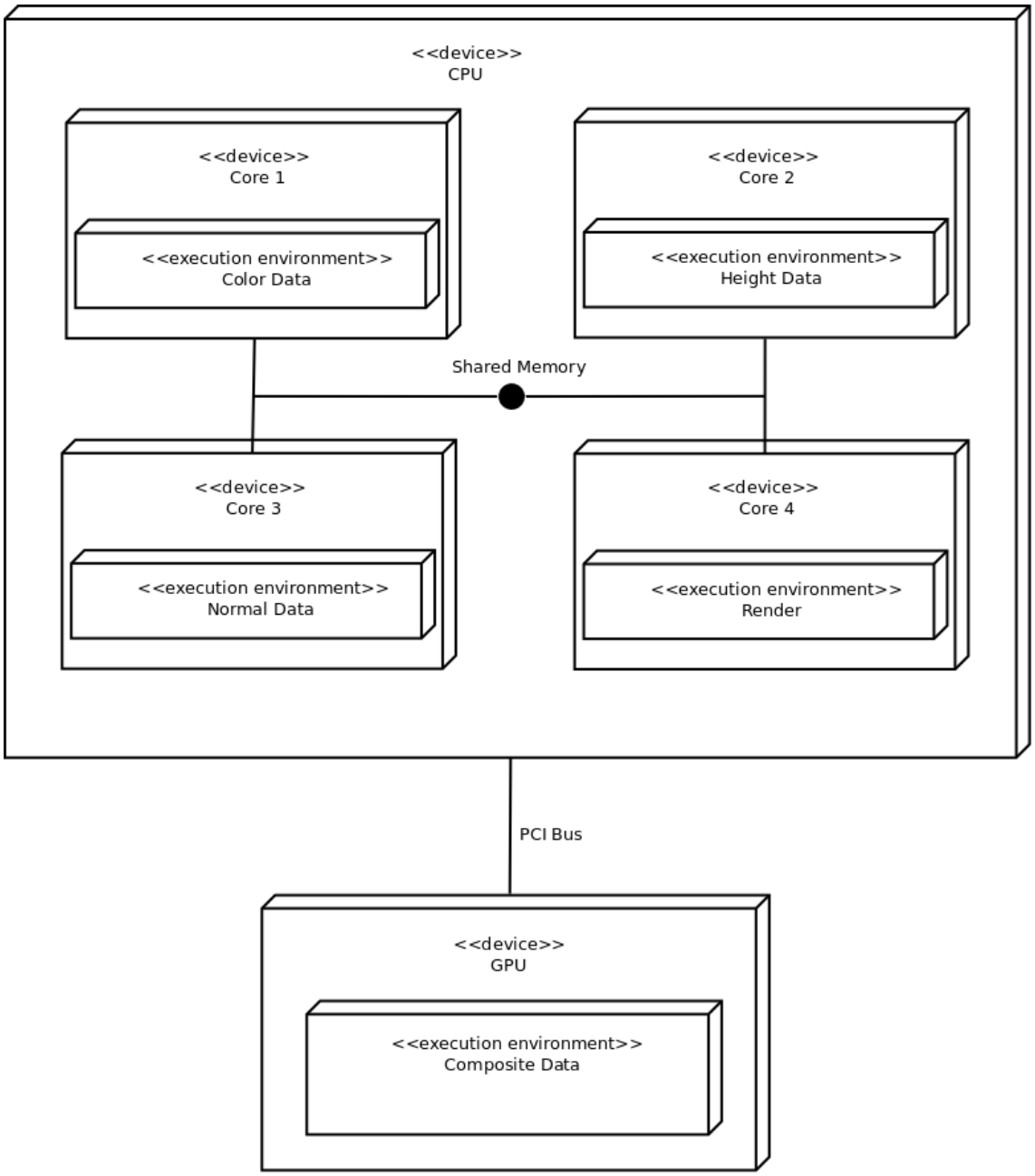

Figure 4.6: Deployment diagram for the system.

Since the data-cacher is designed to handle three separate types of data (color,

height, and normal), a thread is launched for each type to search their respective subtrees individually. Therefore, each data type is processed independently of each other so as to not place a large search overhead on the system. The main thread then takes care of uploading data to the GPU and initiating the composition steps. This layout can be seen in the deployment diagram [3] in Figure 4.6.

For efficiency, each of the search threads should obtain their own individual cores of the CPU. This allows for the system to not interrupt a current search operation for a different data type. Additionally, OpenGL only allows for one rendering context per-thread [2]. Therefore, only one of the threads can interact with the GPU. For this reason, the render thread is selected to upload data to the GPU, not the respective threads themselves.

4.6 Desktop Environment

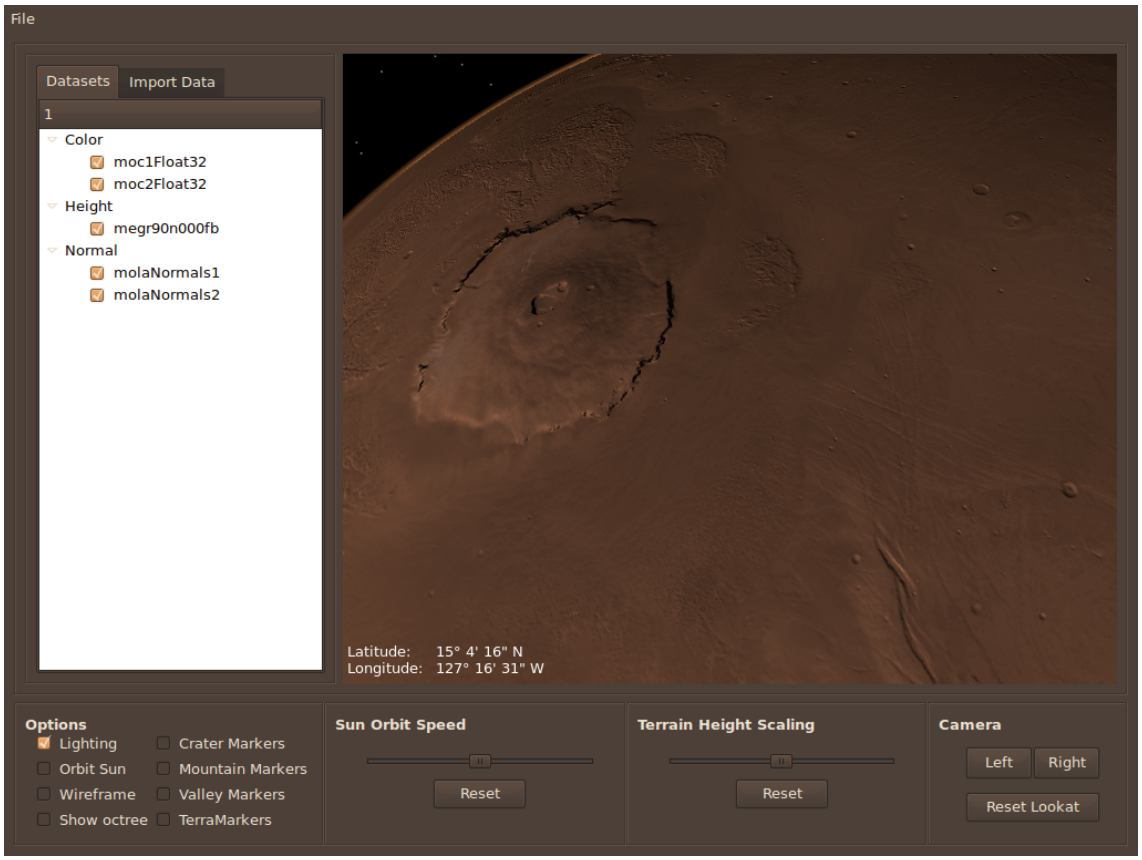

Figure 4.7: The system running in a Qt environment.

To test our algorithm, we chose to implement it in a desktop environment with the use of the Qt programming library [23]. The developed graphical user interface exposes the functionality of the data-caching mechansim along with the Hesperian terrain render library for general use. An example of the running system can be seen in Figure 4.7. This figure shows the currently loaded datasets which are used to render the planet along with their enabled/disabled status.

Chapter 5

Results

5.1 Experimental Method

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

For these tests, the maximum amount of GPU memory is set to 8192×8192 as this is a common graphics card maximum texture size [25]. We chose to use eight datasets to make up the color, height, and normal data for the planet of Mars, detailed in Table 5.1. Additionally, we use the Hesperian [21] terrain renderer and compare the runtimes of our algorithm with the time Hesperian takes to run for similar scenes. The screen size is set to 1280×720 with a grid size of 1280×720.

For the tests, we have selected three separate views of Mars. First, we use a global view which will contain one half of the planet, shown in Figure 5.1. This view of the planet will have a lower LOD because the viewer is farther away and therefore less detail needs to be rendered. Therefore, larger tiles of the datasets will make up the final image. Second, we use a view of Olympus Mons, the largest volcano on Mars and in the solar system, shown in Figure 5.2. As the viewer is close, higher-resolution data will be used for the rendering. Last, we perform a flyby of the equator to show how the data caching mechanism performs while swapping data in and out of GPU memory. This is performed approximately one hundred meters in the air above the

Table 5.1: Information about the datasets used for timing the algorithm, with a total of 5335.39MB of data.

Name	**Scale (m/px)**	**Width**	**Height**	**Size (MB)**
MOC Tile 1	2604.699	4096	4096	191.84
MOC Tile 2	2604.699	4096	4096	191.84
MOLA Heights	1853.000	11520	5760	126.56
MOLA North Heights	115.000	12288	12288	576.09
Victoria Crater Heights	1.011	1279	1694	8.27
MOLA Tile 1 Normals	1852.230	5760	5760	379.73
MOLA Tile 2 Normals	1852.230	5760	5760	379.73
MOLA North Normals 1	115.000	5760	5760	864.13
MOLA North Normals 2	115.000	5760	5760	864.13
MOLA North Normals 3	115.000	5760	5760	864.13
MOLA North Normals 4	115.000	5760	5760	864.13
Victoria Crater Normals	1.011	1279	1694	24.81

surface of the planet. Therefore, higher-resolution data will be swapped in and out of the texture atlas for areas around the viewer while lower-resolution data will be used for views in the far off distance.

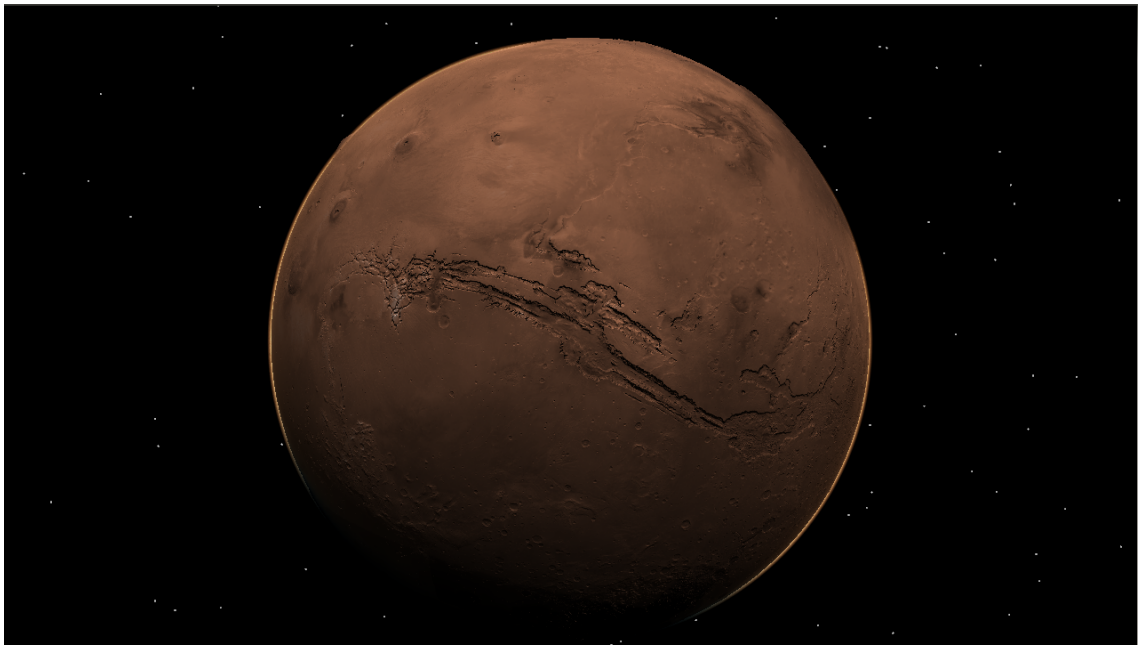

Figure 5.1: Global view of Mars.

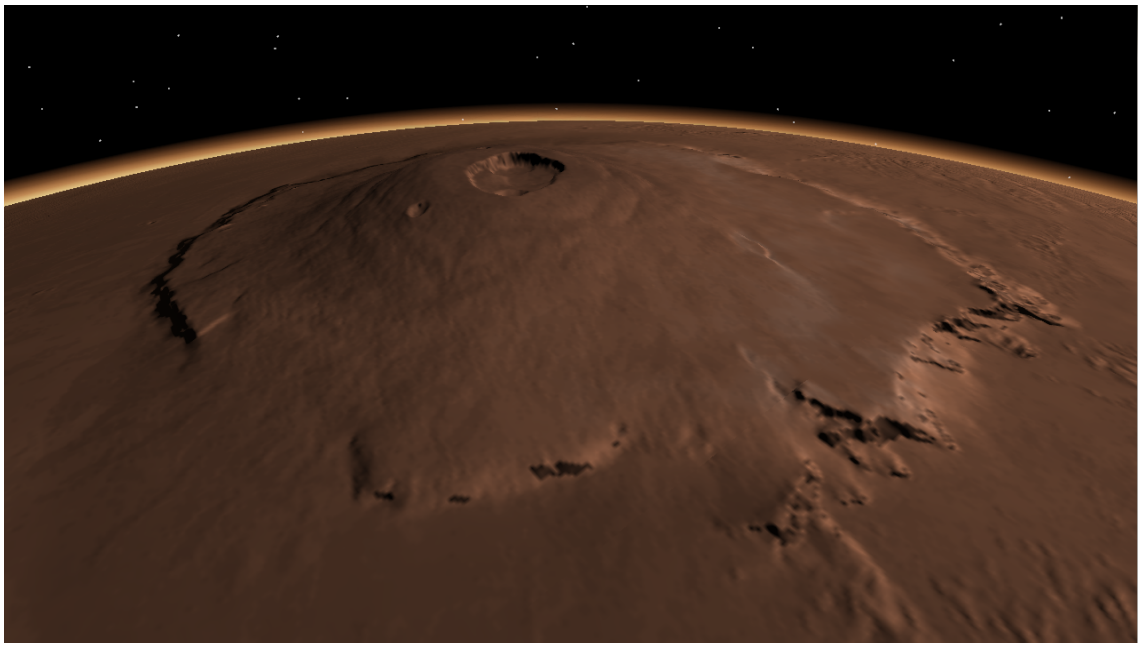

Figure 5.2: Olympus Mons.

5.2 Results and Analysis

We begin our analysis with a measurement of the frames-per-second (FPS) obtained from rendering the three scenes detailed above determined from an average of 10,000 frames. Table 5.2 shows an FPS comparison of the data-cacher along with the original Hesperian terrain renderer. We can see from these results that in a static scene, the data-caching mechanism performs a little faster than the original Hesperian renderer. This is largely due to two factors. First, the data-cacher renders a screen-aligned quad which only covers the region of the dataset while Hesperian render a full-screen quad for each dataset. Therefore, less fragments need to be processed by the GPU with the new method. Second, the camera is static in the world, the data-cacher can detect this and not recomposite the same data each frame. Therefore, the compositing step needs to only happen once per camera movement, taking less time from the terrain rendering process.

In the flyby scene, the data-cacher performs slightly slower than Hesperian. This is because Hesperian always processes each dataset the same per frame, therefore

camera movement does not affect the overall rendering performance. However, the data-cacher must upload new data to the GPU and perform maintenance tasks on the loaded terrain patches so as to not overload the GPU memory. Therefore, this drop in framerates is to be expected as the algorithm becomes limited by swapping terrain patches.

Table 5.2: FPS results with a comparison to Hesperian.

Scene	**Data Cacher**	**Hesperian**	**Percent Different**
Planetary View	127.15	118.04	+7.7%
Olympus Mons	127.36	115.52	+10.2%
Flyby	107.20	111.54	-3.89%

Next, we analyze the timing results of the three main processes of the data-caching algorithm, uploading data to the GPU, compositing the data into images, and terrain patch queue maintenance. To accurately determine how much time is spent in these steps, we chose to time the flyby scene as it represents the worst-case use scenario of the algorithm where data is constantly being streamed from the hard disk to the GPU. We can determine from Table 5.2 that the average time in milliseconds (ms) spent rendering a given frame is 9.32ms. Table 5.3 shows the timing results.

Table 5.3: Timing results of the rendering algorithm.

Process	**Time (ms)**	**Percent of Runtime**
Uploading	0.064	0.68%
Compositing	0.306	3.28%

Therefore, the two main rendering features of our algorithm take up only 4.82% of the overall time to render a single frame. This proves that our algorithm is almost completely decoupled from the terrain rendering and therefore does not dramatically affect the rendering performance.

Additionally, we timed the maintenance step for this same flyby scene at 0.033ms for sorting the data and discarding any old terrain patches to the hard disk. Since this step does not affect the rendering of individual frames, it was not included in the previous table but is important to note as it does affect the overall efficiency of the system.

5.3 Visual Results

Since our data-caching mechanism has been created to display high-detail data at realtime framerates, it is important to show the visual results of the algorithm as well. We start with a comparison of the same scenes as shown in Figure 5.1 and Figure 5.2 as rendered from the original Hesperian implementation [21]. This comparison is made to show the improvement in visual quality that is obtained from using our data-cacher over the previous implementation. A global view and one of Olympus Mons can be see in Figure 5.3 and Figure 5.4 respectively.

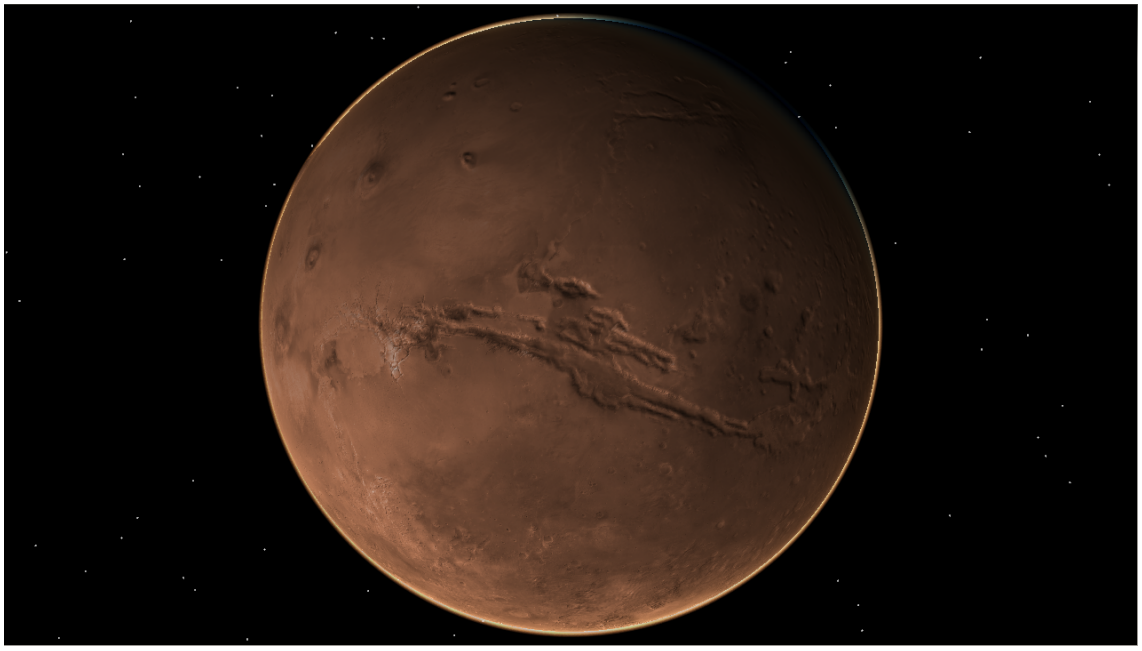

Figure 5.3: Rendering of the planet Mars from Hesperian.

These images can be compared with the higher-resolution images from Figure 5.1

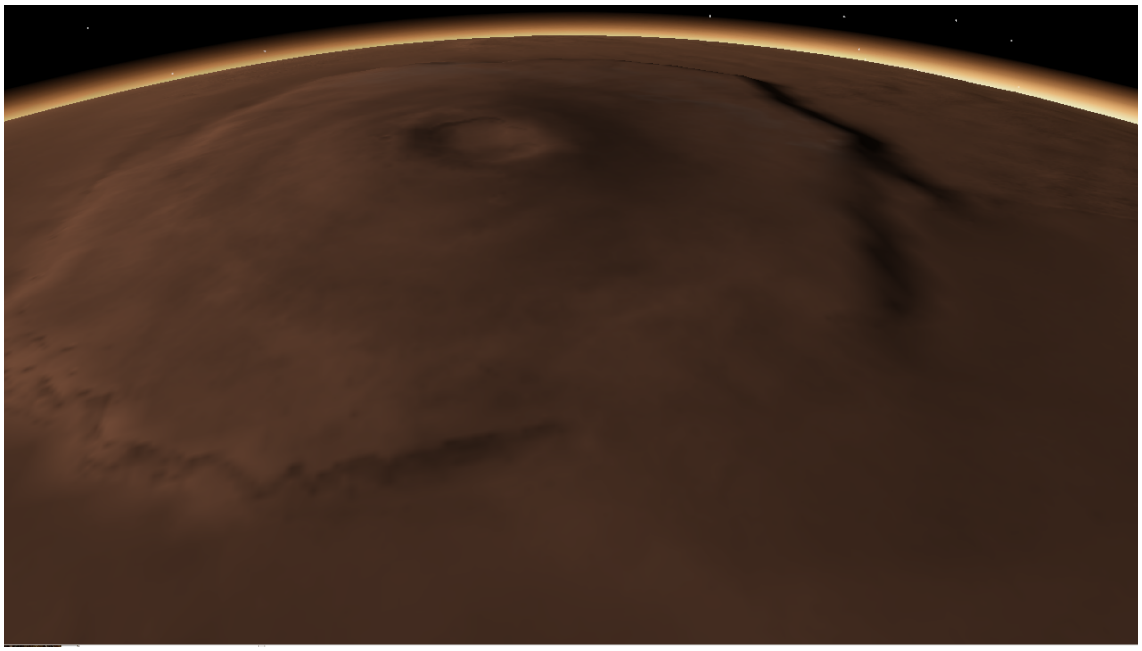

Figure 5.4: Rendering of Olympus Mons from Hesperian

and Figure 5.2 respectively to see the difference in detail. While our algorithm runs at similar framerates, it achieves a higher visual quality because it is capable of using higher-resolution data whereas Hesperian contains no data streaming techniques. Therefore, it can be seen that our enhancement to the original Hesperian terrain rendering library improves both the visual quality and efficiency of the renderer.

Next, we show example images of some high-frequency areas of Mars. Figure 5.5 shows a part of the Mariner Valley, a large canyon near the equator of Mars. Shown in this image is nicely sloped hills which is due to the high-resolution height data used for the planet.

Additionally, an example of some large rock formations near the south pole can be seen in Figure 5.6.

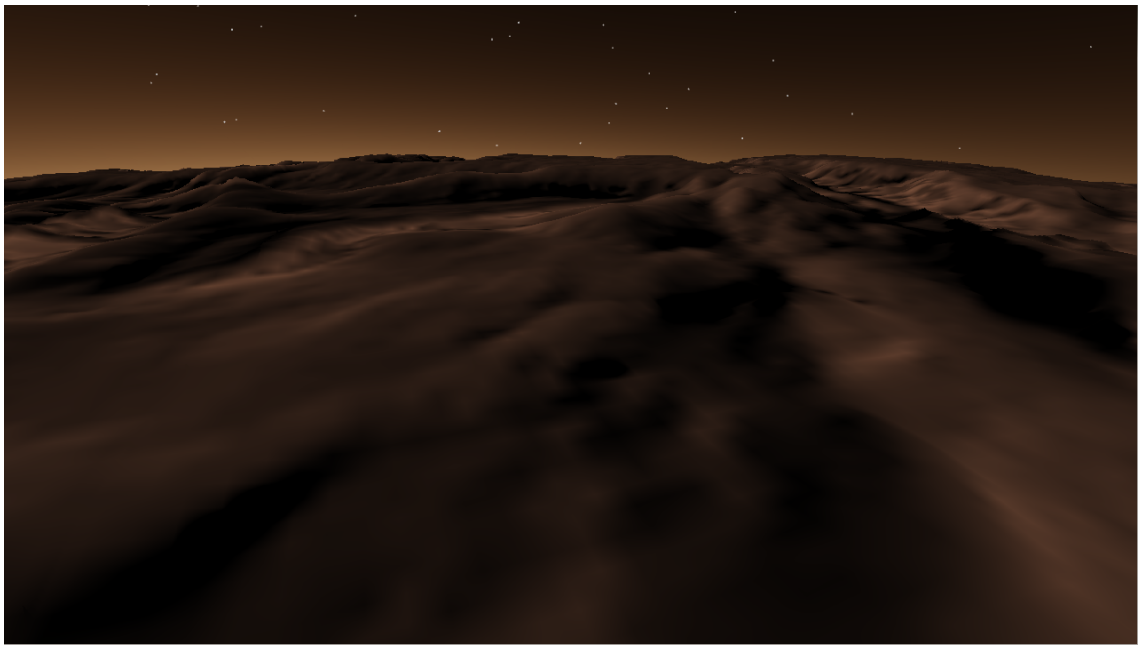

Figure 5.5: Mariner Valley.

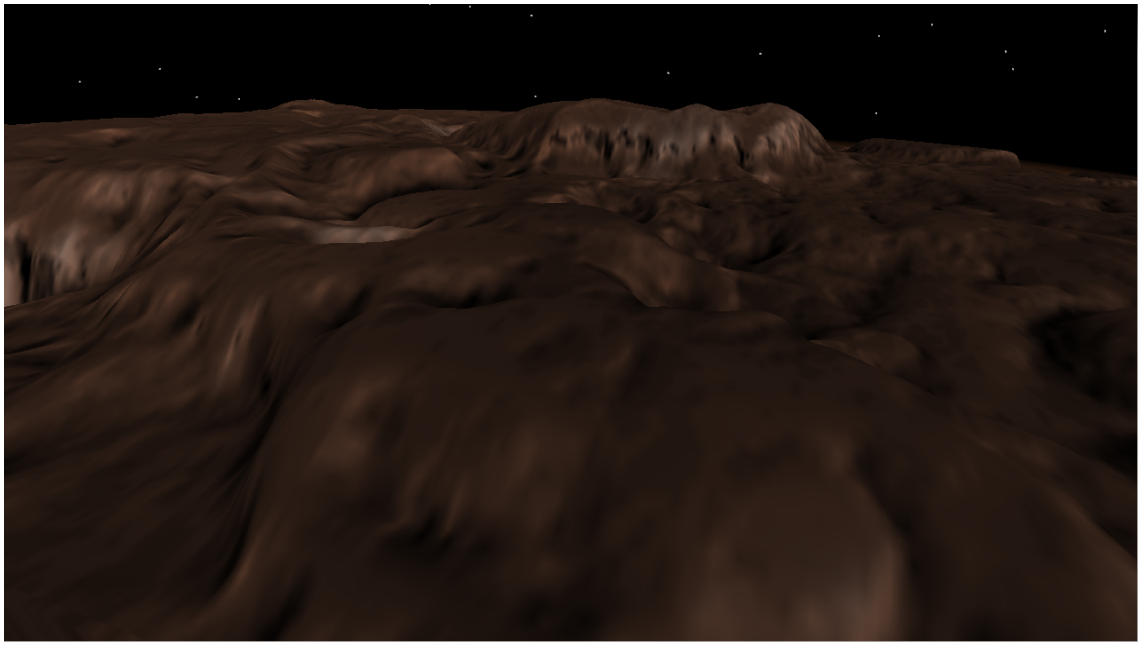

Figure 5.6: Large rock formations near the south pole.

The way our algorithm works does produce a small visual caveat however. As the user moves around the planet, new data is loaded into the atlas and sorted based on its distance to the user, as explained in Section 3.3.3 and Section 3.3.5 respectively. While this process is performed in a background thread, the user may be moving faster than the system can while performing the uploading and maintenance tasks. Therefore, the user may notice areas of the planet where no data is loaded in. An example of this is shown in Figure 5.7.

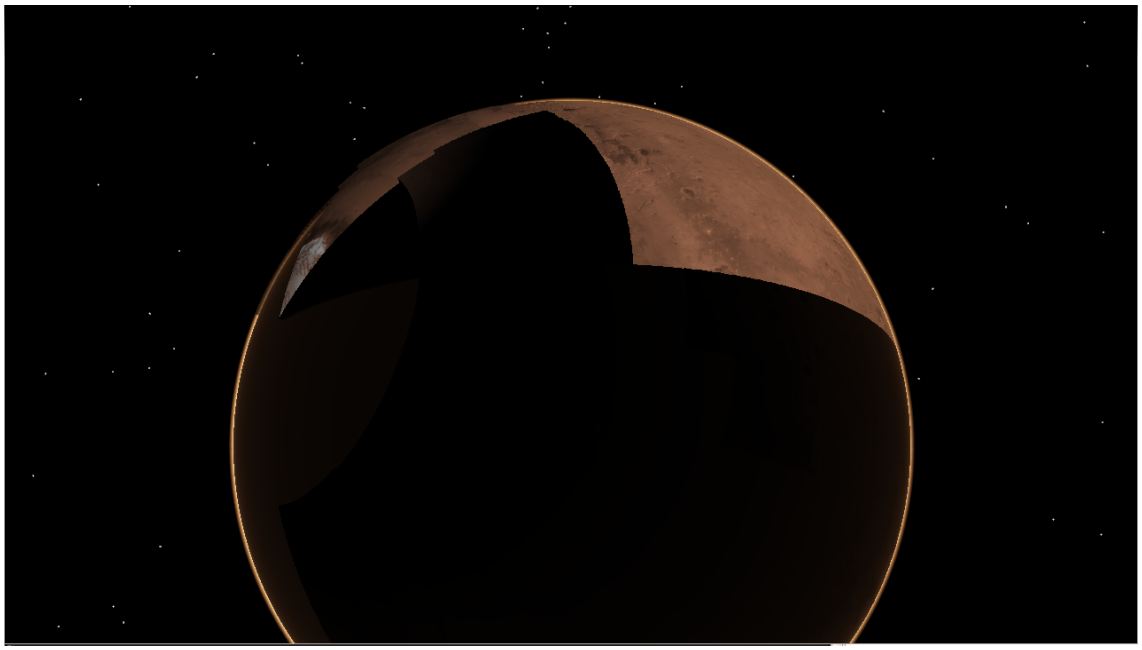

Figure 5.7: The dark areas have not yet been loaded into the texture atlas.

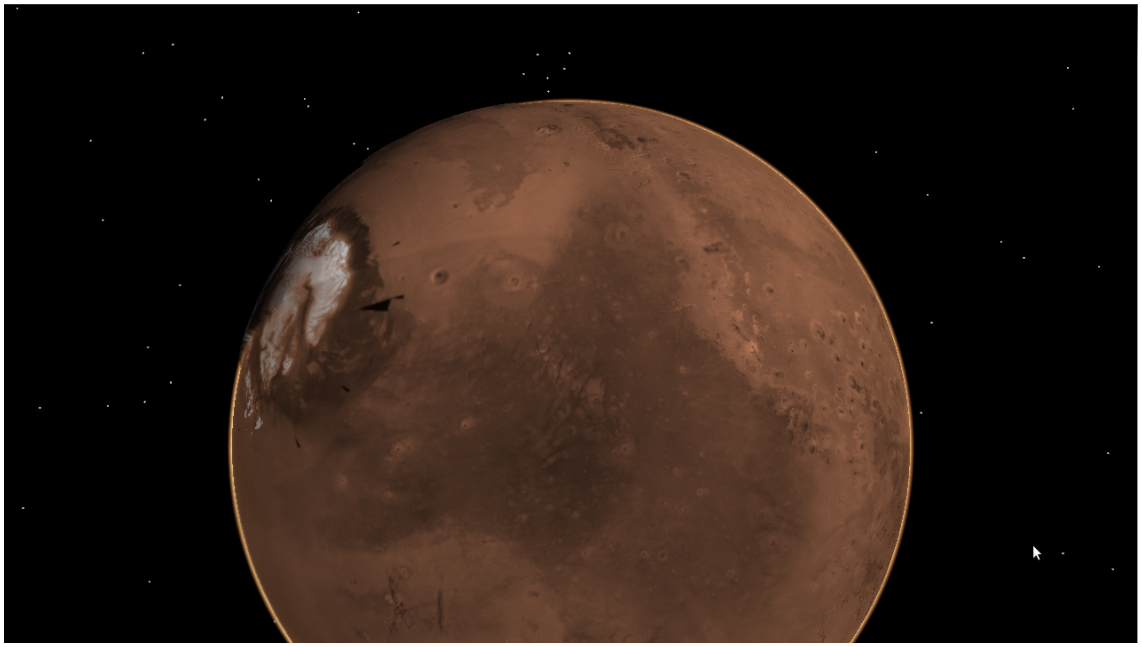

Figure 5.8: The fully loaded planet data.

However, this is corrected by the system very quickly and the full data is loaded after a few short seconds. As the system compensates almost immediately, it should be noted that Figure 5.7 was hard to obtain from the runtime algorithm. Figure 5.8 shows the final image with all loaded data.

Chapter 6

Conclusions and Future Work

The advantage of the data-caching mechanism presented here is that it utilizes both GPU and multi-core processing, allows for new data to be inserted into the data hierarchy at runtime, and allows for easy rendering in a virtual reality environment. As each dataset is uploaded to the GPU as a single point, the massive parallelism of modern GPU architectures gets to address the problem of turning the points into screen-aligned quads and properly texturing them. Using multiple threads, the system can search for new data while the previous data is being used by the renderer, decoupling the rendering and searching operations. Additionally, the user can determine the overall performance of the system by specifying the amount of memory that the loaded datasets can consume on the GPU. If more memory is allowed, less dataset paging needs to occur therefore speeding up the runtime of the algorithm. This process is important to allow so that a given user can fine-tune the application performance for their respective machine.

We have applied common out-of-core data-caching techniques for a planetary rendering system. This was accomplished by placing datasets into a bounding volume hierarchy (BVH) so that they can be efficiently searched through at runtime. We also explained how to composite the datasets on the GPU into one final image for use by the renderer and how to insert new data into the hierarchy at runtime. Lastly, we showed from our results that the data-cacher is almost completely decoupled from the renderer allowing for quick terrain rendering in a real-time environment.

Based on the analysis of our results, there are a few things which could be done

to improve the algorithm. First, the data streaming could benefit from some sort of compression scheme to lower the amount of data to be transfered to the GPU. The compressed data could then be uncompressed on the GPU. Using this method, the bus speed between the system memory and the GPU becomes less of a bottleneck. Second, the data could be laid out on the hard drive in a more cache-friendly way. It is very likely that datasets surrounding a found dataset will also be uploaded to the GPU. Therefore, if they were laid out differently on the hard drive, the system could spend less time reading from different sectors of the hard disk.

In terms of visual results, we pointed out that there are times when the system is loading in data slower than the user is viewing it. This can easily be fixed by loading a small global color map of the planet at runtime. It should be small and low-resolution so it does not adversely affect the rendering time of the rest of the algorithm or take up too much memory. This data can constantly remain in memory and always be rendered in place of no-data regions. This way, the user can still see the unloaded regions of the planet until the new data is ready to go, replacing the low-resolution global map. While the map would probably be visually unappealing due to its low-resolution, it can serve as an easy placeholder so the user does not get confused. This data can be loaded and used similarly to the global height dataset from Section 3.5

A hard requirement of this algorithm is that it needs at least four processor cores to run efficiently. This is due to the fact that the cacher will typically load height, color, and normal data. Therefore, a thread for each of those data types will be launched and searching for new data based on the user's view. A fourth thread is then needed for rendering. Should the algorithm be run on a system with less cores, the CPU will have to reschedule search operations based on the current needs of the system. This could slow down the system dramatically as the threads are not free to search when needed. While this is not an area for future work, it is a hardware requirement that should be explained to any user.

Bibliography

[1] `http://www.opengl.org/documentation/specs/version1.1/glspec1.1/node84.html` (Accessed November 12, 2010).

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

[3] J. Arlow and I. Neustadt. *UML 2 and the Unified Process*. Addison-Wesley, 2nd edition, 2008.

[4] BDAM. `http://vcg.isti.cnr.it/activities/surfacegrevis/bdam/bdam.htm` (Accessed November 8, 2010).

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

[6] CESNET. CAVE to CAVE: Communication in Distributed Virtual Environment. `http://www.cesnet.cz/doc/techzpravy/2008/cave-to-cave/` (Accessed November 20, 2010).

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

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

[9] J. M. Daughtry, U. Farooq, J. Stylos, and B. A. Myers. Api usability: Chi'2009 special interest group meeting. In *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, CHI '09, pages 2771–2774, New York, NY, USA, 2009. ACM.

[10] W. de Boer. Fast terrain rendering using geometical mipmapping. October 2000. `http://www.connectii.net/emersion`.

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

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

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

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

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

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

[17] T. Lauritsen and S. Nielsen. Rendering Very Large, Very Detailed Terrains, 2005.

[18] C. Lin and L. Snyder. *Principles of Parallel Programming*. Addison-Wesley, 1st edition, 2009.

[19] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *IEEE Visualization 2001*, October 2001.

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

[21] J. D. Mahsman. Projective grid mapping for planetary terrain. Master's thesis, Department of Computer Science and Engineering, University of Nevada, Reno, December 2010.

[22] NASA. Mars - images of mars. `http://www.nasa.gov/mission_pages/mars/images/index.html` (Accessed December 10, 2010).

[23] nokia. Qt - a cross-platform application and ui framework. `http://qt.nokia.com/products/`.

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

[25] Nvidia. GeForce GTX 580. `http://www.nvidia.com/object/product-geforce-gtx-580-us.html` (Accessed November 13, 2010).

[26] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 7th edition, 2010.

[27] R. J. Rost. *OpenGL Shading Language*. Addison-Wesley, 2nd edition, 2008.

[28] S. S. Somé. Supporting use case based requirements engineering. *Inf. Softw. Technol.*, 48:43–58, January 2006.

[29] I. Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2010.

[30] A. Sutcliffe, B. Gault, T. Fernando, and K. Tan. Investigating interaction in cave virtual environments. *ACM Trans. Comput.-Hum. Interact.*, 13:235–267, June 2006.

[31] R. N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 1st edition, 2010.