SOFTWARE DEVELOPMENT ASPECTS OF OUT-OF-CORE DATA MANAGEMENT FOR PLANETARY TERRAIN

Cody J. White, Sergiu M. Dascalu and Frederick C. Harris, Jr.
University of Nevada, Reno, U.S.A.

Keywords: Terrain-rendering, Data management, Out-of-core, GPU, Software requirements, Software design.

Abstract: Rendering terrain on a planetary scale can quickly become a large problem. Many challenges arise when attempting to render terrain over a spherical body as well as deal with the large amount of data that needs to be used to accurately display the terrain of a planet. Most research in the area of terrain rendering is specific to a given region of a planet, therefore needing few datasets for a proper rendering. However, since planets are made up of larger areas, a different approach needs to be taken in order to display high-detail terrain around a viewer while sorting through the large amount 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 must be created. Therefore, we present the software design for a data caching mechanism which can efficiently swap only the data around a viewer into and out of the GPU memory in real-time. We also present a prototype of the software which achieves efficient framerates for high-quality views of a planet's surface while minimizing the time it takes to find data centered around a viewer and display it to the screen.

1 INTRODUCTION

Terrain rendering has been at the forefront of research for many years, especially in the fields of video games, scientific visualization, and training simulations. While many problems have been solved, little has been done to address the issues surrounding rendering terrain on a planetary-scale. These complexities arise from both the shape of a planet as well as the amount of data that must be used in order to accurately render the planet in detail. While there have been many solutions for rendering planetary bodies, (Mahsman, 2010), (Cignoni et al., 2003), the problem of dealing with large amounts of data still needs to be fully addressed.

A planet's surface is typically made up of many datasets which describe the various geographical areas, ranging form very small in size to terabytes of information. Fortunately, there has been a large amount of research in dealing with extremely large datasets for terrain rendering (Cignoni et al., 2003), (Dick et al., 2009), (Kooima et al., 2009). As this is the first step to generating realistic terrain, it has received much attention. Typically, the data is organized into a spatial hierarchy and then chosen for rendering based on a user-specified search criteria. Using this approach, only the data surrounding a user needs to be in memory while the rest can safely remain on the hard drive until it is needed. While this process is extremely helpful in handling large datasets efficiently, it does not solve the problem of multiple datasets needing to be considered. Therefore, an extension to this approach must be devised.

As planets are of a fairly predictable shape, a bounding box can be constructed which tightly contains them. Using this information, it can be possible to construct a spatial subdivision hierarchy which covers the region of the entire planet. This hierarchy can then contain information for the system to determine which datasets that exist on the planet are currently in view. Using this approach, the use of a large amount of datasets is simple to deal with. Additionally, inserting new data into the hierarchy becomes a trivial operation. Once the hierarchy has been created, it can exist in the main memory without any data loaded to simply determine the visibility of datasets.

As a core requirement of terrain renderers, the algorithm should work in real time. To achieve this result, the data cacher should take advantage of both CPU and graphics processing unit (GPU) parallel processing. Much of the searching and data swapping can occur in parallel, freeing up system resources for the terrain renderer being used. Since the terrain renderer can use data already loaded into mem-

J. White C., M. Dascalu S., C. Harris and Jr. F. (2011).
SOFTWARE DEVELOPMENT ASPECTS OF OUT-OF-CORE DATA MANAGEMENT FOR PLANETARY TERRAIN.
In *Proceedings of the 6th International Conference on Software and Database Technologies*, pages 185-191
DOI: 10.5220/0003509201850191

ory, the data cacher can work independently of the renderer for its searching operations. The GPU can also be used to perform composition of terrain data into a final image for use in the generation of a three-dimensional mesh, similar to (Kooima et al., 2009).

We present the software development aspects for the creation of a data caching system as described in the thesis *Out-of-Core Data Management for Planetary Terrain* (White, 2011). This system processes the many datasets that compose a full planet and efficiently swaps them in and out of GPU memory in real time. As the data caching system should not interfere with the rendering, the system utilizes parallel processing to resolve this issue. Additionally, the system supports the addition of new datasets at runtime to allow the user to render data which is not part of the pre-built data cache. Figure 1 shows a resulting frame from our proposed algorithm and system.

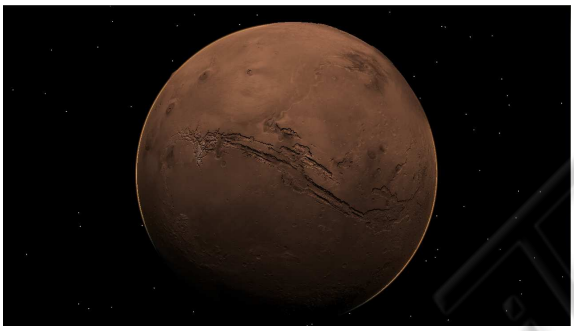

Figure 1: Global view of the planet Mars.

In this paper, we start with a brief survey of related work (Section 2). Our software has been designed with several considerations in mind which strive to hide the complexities of data caching from the user (Section 3). Instead of creating a new terrain renderer to test the software, we integrated our algorithm with the Hesperian terrain rendering library (Mahsman, 2010). Upon testing our software with the terrain renderer, we've seen a noticeable improvement in both the visual quality of the rendered terrain as well as the overall efficiency of the system (Section 4). Additionally, we present some ideas for a future expansion of the software system (Section 5).

2 RELATED WORK

A common approach to breaking datasets into smaller, more memory cohesive chunks is to subdivide them into a quadtree hierarchy (Lindstrom et al., 1996), (de Boer, 2000), (Lindstrom and Pascucci, 2001). This type of structure is used because it splits data into four equal-sized chunks per node until a predefined threshold has been met whereupon mipmapping can occur. 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 level-of-detail (LOD) for the given dataset.

A simple algorithm for mapping many different datasets together is known as deferred texturing (Kooima et al., 2009). Using this, a texture atlas can be created which stores all of the loaded datasets for access by the GPU. This method easily relates to planetary data as each chunk of the atlas can represent a geographical region of the visible area. While this helps to solve the issue of compositing multiple datasets together on a planetary scale, the system implemented in (Kooima et al., 2009) is unable of rendering without a search pass through their hierarchies. Therefore, the overall speed of the system is dependant on the efficiency of their data swapping mechanism. This limitation forces the terrain renderer to work at the speed of the hard drive and memory bus, which restricts how often a frame can be rendered. However, a data-caching system should not inhibit the renderer's performance. Therefore, it is an important aspect for our system to decouple the data caching and rendering through the use of CPU parallel processing. Additionally, a generic data cacher should be independent of the type of renderer being used, be it ray or triangle-based. Our system then, has been designed with this constraint in mind.

3 SOFTWARE DEVELOPMENT

In order to accurately define our system, we must first determine the functional and non-functional requirements (Sommerville, 2010) (Pressman, 2010) (Section 3.1). Additionally, it is imperative to understand the user's needs for the system (Section 3.2).

3.1 Requirements

Our proposed system 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 (Lin and Snyder, 2009) and rendering context-safe. Additionally, we make use of cross-platform APIs (Daughtry et al., 2009) such as OpenGL (Angel, 2008) and GDAL (GDAL,

2011). 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.

In this NASA funded project, 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, color, or normal data, can be obtained directly from NASA (NASA, 1976), (NASA, JPL, and University of Arizona, 2011), (NASA Goddard Space Flight Center, 1996). 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, a composition algorithm is used 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. Since we do not want the data cacher to inhibit the realistic terrain rendering, we utilize the multi-core power of modern CPUs and push all searching, uploading of patch data, and addition of new data to separate threads.

Using common projection equations (Eliason, 2007) we are able to place any dataset in the proper geographic location. Currently, our implementation supports both equirectangular and polar stereographic projections. However, more projection types could be trivially added. As different data can be stored in these projections, it is imperative to use the proper projection when transforming the projection coordinates into texture coordinates or the image will have continuity problems related to an improper projection.

These functional and non-functional requirements are listed in Table 1 and Table 2 respectively.

Table 1: Functional requirements.

F01	The library will read a standard data format.
F02	The library will allow variable patch sizes.
F03	The library will allow for new data.
F04	The library will composite overlapped data.
F05	The library will allow variable LOD error.
F06	The library will allow scalable memory use.
F07	The library will preprocess terrain data.
F08	The library will select the proper LOD.
F09	The library will display datasets correctly
F10	The library will use threads.
F11	The library will make use of the GPU.
F12	The library will provide a simple interface.

Table 2: Non-functional requirements.

N01	The system will be implemented as a library
N02	The library will be implemented using C++.
N03	The library will be thread safe.
N04	The library will be rendering context safe.
N05	The library will use OpenGL.
N06	The library will use GLSL.
N07	The library will use GDAL

3.2 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 its 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 (Lindstrom et al., 1996), the dataset bounding volume hierarchy (BVH), and pre-determining dataset errors. Once the data is processed, it is written to the hard drive and stored for later use. These written files then make up the data cache. This functionality is simple to use in our library as the interface can accept a list of datasets to preprocess. Once this step has been completed, the renderer is able to use the data cache for rendering purposes.

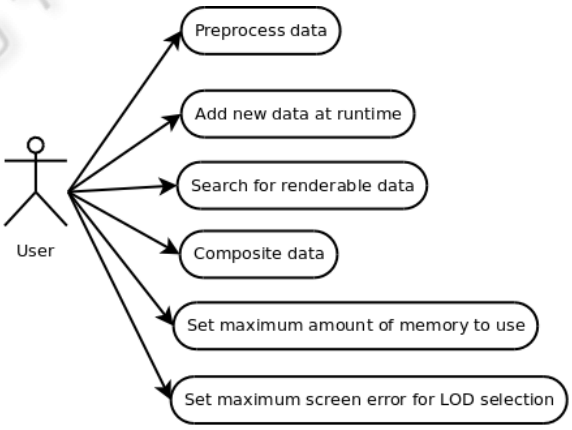

Figure 2: Use cases for the data-caching library.

At runtime, the terrain renderer can add new data to the data cache which can either be saved to the data cache 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 (Somé, 2006) are shown in Figure 2.

3.3 Classes

As we have strived to maintain usability and simplicity in the use of our implementation, we have written the code in as few classes as possible. For easy integration, the user only interacts with one class which is used to hide the complexities of the system beneath it. An overview of these classes can be seen in the class diagram (Taylor et al., 2010) presented in Figure 3.

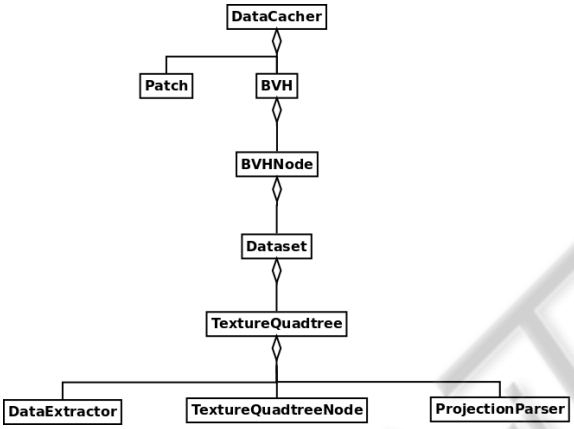

Figure 3: Class diagram for the data-caching library.

The `DataCacher` class acts as an interface into the rest of the system. With this 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. We have developed the `ProjectionParser` class which obtains all of the necessary projection information for a particular dataset.

To simplify the code, all GDAL commands are contained within the `DataExtractor` class. Using this class, the system can determine projection information, obtain regions of pixel data, and calculate the projection coordinates of a dataset. In order to support rendering-context safety, we allow for multiple terrain patch queues to exist (one per graphics context). Therefore, different displays that are looking in different directions in a 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 which segregates the terrain data for each context into separate instances of any class that needs to be context-safe.

Outside of providing just an interface, the `DataCacher` class also takes care of maintaining the queue of visible terrain patches 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. This class also uploads all patches to the GPU by the creation of a texture atlas, similar to (Kooima et al., 2009). While the atlas is being uploaded, a legend is also being created which details where each dataset resides in the atlas, as well as per dataset information required by the GPU for composition. As a final step, the center of each dataset in world coordinates is saved in a vertex buffer object (VBO) for rendering. Once the atlas has been uploaded, the GPU takes over for the final steps of the implementation.

3.4 GPU

Using programmable shaders (Rost, 2008), 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.

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 (Arlow and Neustadt, 2005) depicting the flow of events in this algorithm is shown in Figure 4.

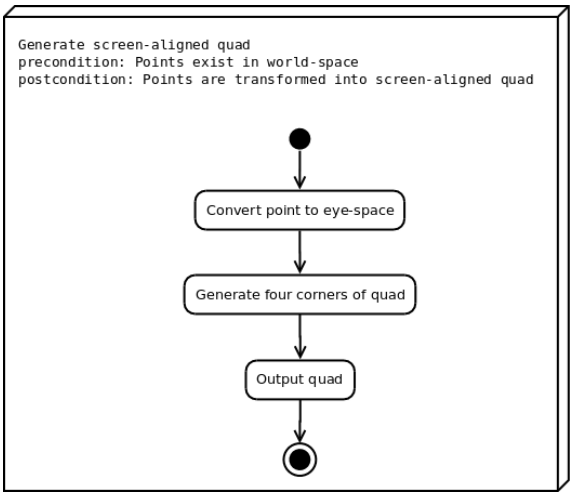

Figure 4: Generation of screen-aligned quads.

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

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 output quad gets processed as shown in Figure 5.

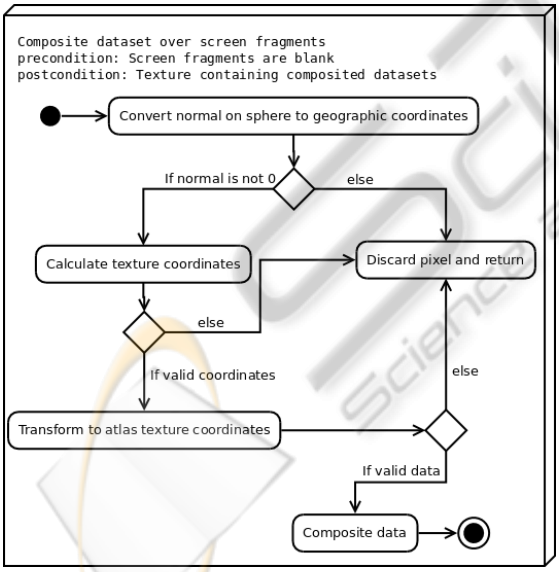

Figure 5: Data composition into final the texture.

3.5 Deployment

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 (Arlow and Neustadt, 2005) presented in Figure 6.

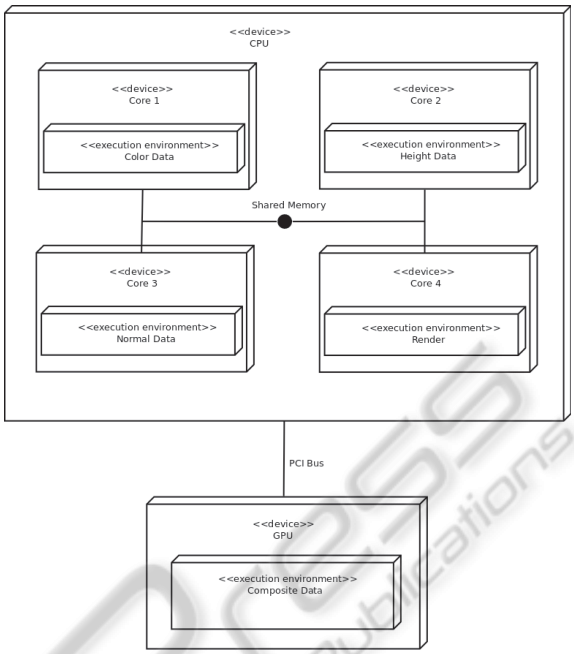

Figure 6: Deployment diagram for the system.

4 RESULTS

To test the functionality of the system, we implemented the class structure from Section 3.3 along with the Hesperian terrain renderer (Mahsman, 2010). To accurately test our solution, we implemented it in a virtual reality (VR) system via Hydra (Hydra, 2010) as well as on a desktop platform using the Qt development environment (Nokia, 2011). A resulting image from the desktop version can be seen in Figure 7. From the desktop view, a user is able to move about the planet, adjust the datasets in use by the system, add new data, and edit lighting and scaling factors. Further figures are from the VR version.

4.1 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.

For all tests performed, the system has loaded 5.5GB of terrain data for use in the visualization.

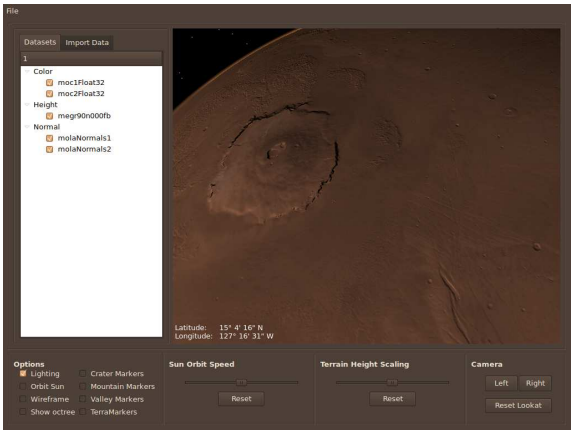

Figure 7: Desktop version of the implemented system.

Since Hesperian is designed to render the planet Mars, all resulting images shown are of the Martian surface.

4.2 Results and Analysis

To determine if the implemented system was successful, we performed numerous tests against a base version of the Hesperian terrain rendering library. This base version supports no out-of-core data streaming mechanism and therefore can only use data that fits within system memory. Visually, our algorithm performs much better as it is capable of handling high-quality data, as can be seen in the comparison between Figure 9 and Figure 8.

Figure 8: Olympus Mons from Hesperian.

Outside of visual results, it was determined that our system ran nearly 10% faster than the base Hesperian implementation (White, 2011). This is largely due the fact that our algorithm has knowledge of the datasets and can create tightly-fitting bounding geometry around them in the geometry shader whereas Hesperian must render a full-screen quadric for every dataset during the composition pass of the rendering algorithm. Therefore, less screen pixels need to be executed by the GPU from our implementation, decreasing the runtime of the composition step.

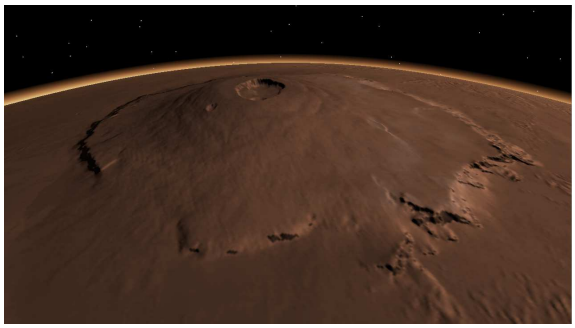

Figure 9: Olympus Mons from our algorithm.

In order to prove that the parallelization of the data caching algorithm was effective, we needed to determine that the implemented system was decoupled from the terrain renderer. From our analysis, we calculated that approximately 4% of the overall execution time for each frame of the visualization is spent dealing with our data caching system. Therefore, we can show that our system is in fact decoupled from the terrain renderer as the remaining 96% of the execution time is spent in the Hesperian system rendering to the screen. From this, we can determine that the system is successful in not forcing a constraint on the overall rendering system performance from searching the dataset hierarchies and uploading data to the GPU for insertion into the texture atlas.

5 CONCLUSIONS

We have presented the software design for an out-of-core data caching mechanism designed to work for a planetary terrain renderer. By breaking the problem down into manageable classes, we have created a simple to maintain and use system which can easily be integrated into a terrain rendering system which uses either a triangle or a ray-based terrain generation approach. In utilizing the GPU, the system is able to approach efficient run time speeds. Additionally, the design of a GPU algorithm means that this system will need no modification for future hardware as GPU architecture improves due to the fact that the GPU code is written in GLSL, which automatically scales with the number of cores present (Rost, 2008).

From the analysis of our results, we have proven that the system is both efficient and decoupled from the terrain renderer. Therefore, if the system were to be integrated with other terrain renderers, no noticeable drops in framerates should occur. We have achieved this through the parallelization of our search

algorithms as well as a joint CPU-GPU design which offloads heavy work onto the GPU.

To further improve the system, a data compression system could be implemented to allow for compressed terrain data to exist on the hard drive. Using this approach, data is compressed by the terrain preprocessing step and stored for future searches. Once the data is uploaded to the GPU, it is uncompressed into its original form for use. The performance benefit from this method is directly related to the amount of data being read from the hard drive and to the GPU.

ACKNOWLEDGEMENTS

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

REFERENCES

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

Arlow, J. and Neustadt, I. (2005). *UML 2 and the Unified Process*. Addison-Wesley, 2nd edition.

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

Daughtry, J. M., Farooq, U., Stylos, J., and Myers, B. A. (2009). API usability: CHI'2009 special interest group meeting. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems*, CHI '09, pages 2771–2774, New York, NY, USA. ACM.

de Boer, W. (2000). Fast terrain rendering using geometical mipmapping. http://www.connectii.net/emersion.

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

Eliason, E. (2007). Hirise catalog. http://hirise.lpl.arizona.edu/PDS/CATALOG/DSMAP.CAT (Accessed July 21, 2010).

GDAL (2011). GDAL - Geospatial Data Abstraction Library. http://www.gdal.org (Accessed July 21, 2010).

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

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

Lin, C. and Snyder, L. (2009). *Principles of Parallel Programming*. Addison-Wesley, 1st edition.

Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N., and Turner, G. A. (1996). Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 109–118. ACM.

Lindstrom, P. and Pascucci, V. (2001). Visualization of large terrains made easy. In *IEEE Visualization 2001*.

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

NASA (1976). Mars - images of mars. http://www.nasa.gov/mission_pages/mars/images/index.html (Accessed December 10, 2010).

NASA Goddard Space Flight Center (1996). The Mars Orbiter Laster Altimiter. http://mola.gsfc.nasa.gov (Accessed April 21, 2010).

NASA, JPL, and University of Arizona (2011). HiRISE: High Resolution Imaging Science Experiment. http://hirise.lpl.arizona.edu (Accessed July 21, 2010).

Nokia (2011). Qt - a cross-platform application and UI framework. http://qt.nokia.com/products/.

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

Rost, R. J. (2008). *OpenGL Shading Language*. Addison-Wesley, 2nd edition.

Somé, S. S. (2006). Supporting use case based requirements engineering. *Inf. Softw. Technol.*, 48:43–58.

Sommerville, I. (2010). *Software Engineering*. Addison-Wesley, 9th edition.

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

White, C. J. (2011). Out-of-core data management for planetary terrain. Master's thesis, Department of Computer Science and Engineering, University of Nevada, Reno.