Visualizing Mars in Java3D

James A. Juett
Department of Math, Computer Science, and Physics
Wartburg College
Waverly, Iowa 50677
james.juett@gmail.com

Abstract

Modeling planetary landscapes provides a uniquely promising application area for computer aided visualization. Furthermore, an increasing amount of planetary data from instruments like the Mars Orbiting Laser Altimeter (MOLA) is publicly available via the World Wide Web. In the particular case of Mars, high resolution topographic and image maps are available, and creating an interactive and immersive planetary visualization tool is a challenging, yet straightforward application of these datasets.

This project tests the aptitude of the Java3D graphics framework for planetary visualization. The author's development such an application serves as a case study, especially in the context of an undergraduate project. The final application allows users to "fly" over the landscape of Mars and performs real time level of detail adjustments to the landscape. Alternately, the application may also produce videos through prerecorded paths over the landscape. The application also supports stereoscopic and head tracked viewing for virtual reality environments.

1 Introduction

NASA'a Mars Global Surveyor satellite, launched in 1996, provided scientists with many types of valuable data over the course of its ten year mission. Visualization is crucial in order to make the plethora of data more accessible and glean new information and understanding from the information it represents. In particular, computer aided visualization is a useful tool for expressing this data in a format that is both amenable and intriguing to human viewers.

This is especially true for altimetry data collected by the Mars Orbiting Laser Altimeter (MOLA). The data has been processed by NASA and elegantly organized into Mission Experiment Gridded Data Records (MEGDRs)[1]. Because the MEGDR structure is very well organized, it is straightforward to use the data as input to a number of visualization applications. Possible applications include topographic mapping, watershed analysis, and 3D modeling.

This project focused specifically on creating an application to provide tools for interactive, three-dimensional visualization of planetary landscapes. These visualizations are designed to be accurate, immersive, and to be used in a variety of settings from education to entertainment. Additionally, the final application is set up for display in a passive stereoscopic virtual reality environment and supports head-tracking for truly immersive visualizations.

Choosing a graphics framework on which to build this application was an interesting challenge. While a framework with a relatively steep learning curve was desired for a student researcher, the significant size of the visualization makes efficiency and computing power an issue. The Java3D API [2] was chosen, and a major goal of this application has been to serve as a case study into the utility of Java3D for large-scale planetary visualizations.

2 Java3D

2.1 Exploring the Boundary

Traditionally, there exists a separation between high and low level graphics frameworks. High level graphics frameworks are extremely useful for rapidly creating visualizations. For this reason, they excel as a vehicle to visualize other concepts in educational settings. Low level frameworks, on the other hand, provide a great deal more flexibility and efficiency but often require more time and/or expertise to use.

Java3D lies on the boundary between these two extremes. Java3D presents a friendly scene graph oriented structure while still being efficient enough to support large scale visualizations. The scene graph offers a more intuitive model to beginning graphics

programmers, namely students, than does the state machine design of low level graphics programming. Most people would much more readily describe and think of the world around them in terms of shapes and/or functional objects than in terms of basic polygons, colors, or other primitive elements.

Exploring the boundary between high level and low level graphics frameworks also allows student researchers an experience unique from exploring one of the extremes. Working with a framework like Java3D allows one to experience the best and worst of both worlds and gain a greater appreciation for the difference between the two. By exploring the boundary, one may quickly determine whether a lower or higher level approach is needed or if a compromise like Java3D suits a particular task.

2.2 Java

A benefit of choosing Java3D for this visualization project and for others is the ease of integration with other Java frameworks and APIs, especially the Swing GUI toolkit [3]. The author's familiarity with Java allowed the focus of the project to be on the planetary visualization aspects of the application rather than on learning a new language, GUI toolkit, and/or other necessary tools.

Java's built-in threading architecture [4] also benefits this project. Concurrent tasks are handled in a simple and standard way that does not require or depend upon an outside library. Although several excellent third-party threading libraries exist for a number of languages, the simplicity of Java's threading architecture again allowed more focus to be devoted to the visualization aspects of the application.

2.3 Rendering

Java3D offers three different rendering modes around which an application may be structured. While an application usually adopts one mode as a standard model, the modes are not strictly mutually exclusive. In the planetary visualization application, most of the rendering work is done under compiled-retained mode, but some specialized parts of the application rely on the flexibility of immediate mode.

2.3.1 Immediate Mode

While rendering in pure immediate mode [2], Java3D resembles a low-level graphics framework. In this mode, no automatic rendering takes place. Java3D's geometry and other objects are still available to facilitate rendering, but the programmer directly controls all steps of the process. This mode offers the greatest flexibility at the price of reduced rendering speed. Immediate mode may also be mixed with other modes in

situations that require very fine control for only a small part of the rendering process.

2.3.2 Retained Mode

Retained mode [2] shifts the programmer's focus to the scene graph while the Java3D renderer handles the rendering process. While this sacrifices fine control over rendering, it greatly simplifies the creation of a virtual world. An application simply creates a scene graph, and the Java3D renderer goes to work. This mode actually allows Java3D to make a number of automatic optimizations to the scene graph as it is constructed and thus often enjoys increased efficiency over immediate mode rendering.

2.3.3 Compiled-Retained Mode

Compiled-Retained mode [2] is very similar to retained mode in that Java3D assumes control over lower level rendering activities. Compiled-retained mode adds more optimizations by allowing parts of a scene graph to be compiled by Java3D for more efficient rendering. Once part of the scene graph is compiled, there are few ways in which it may be modified. This allows Java3D to reorganize those objects so they may be most efficiently rendered. Specific care must be taken, however, to specify those parts of the scene graph that may need to be modified by the application.

The bulk of rendering by the planetary visualization application developed during this project falls under compiled-retained mode. The increased efficiency in rendering compiled objects allows relatively high frame rates to be achieved when viewing a static landscape. However, this also means accommodating level of detail changes to the landscape takes more time. Balancing rendering efficiency and scene graph flexibility has proven challenging because the needs of the application vary widely as it is used. For example, a user might be exploring a small, specific area on the planet's surface and suddenly decide to zoom out to see a much wider view of the landscape. One situation calls for efficient rendering of a detailed set of landscape already loaded into the program while the other requires several pieces of landscape to be swapped in and out.

2.4 View Model

In the Java3D scene graph, nodes representing virtual objects are kept separate from those representing the viewing architecture. Each scene graph contains a content subgraph that holds anything to be displayed in the virtual world. A different subgraph contains objects that specify what sort of view a user should have into the virtual world. For example, the ViewPlatform class specifies the user's location in the virtual world, but does not contain anything that needs to be rendered.

Java3D's view model, described in [2], differs considerably from traditional camera-based view models. While traditional camera-based control is still available in Java3D, the focus of Java3D's view model is on the relationship between the virtual world and the physical world in which the user resides. Since many virtual reality environments are based on this relationship, Java3D is able to provide straightforward built-in support for virtual reality techniques like stereoscopic viewing and head tracking.

3 Planetary Visualization Case Study

3.1 Goals and Rationale

The purpose of this project has been to create an interactive and immersive planetary visualization tool with Java3D. In so doing, the project serves as a case study into the utility of Java3D for creating large scale visualizations, especially in the context of student projects. The application has also served as an independent research project and a medium for personal exploration and experimentation by the author.

3.2 Application Design

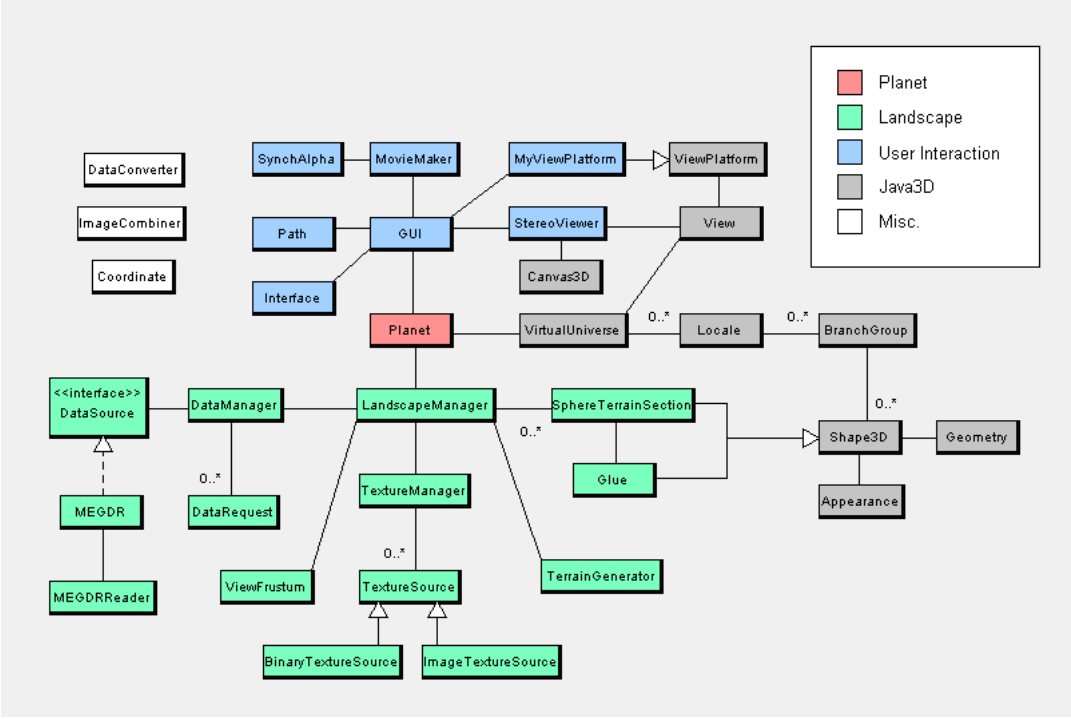

Figure 1: Simplified Architecture.

3.2.1 Data

NASA has made available the altimetry data collected by the MOLA in the form of Mission Experiment Gridded Data Records (MEGDRs) [1]. Each MEGDR constitutes a topographic map containing height values corresponding to particular latitudes and longitudes on the planet's surface. MEGDRs are available at resolutions of 4, 16, 32, 64, or 128 height values per degree. To put this in perspective, each height or "pixel" in the 128 resolution MEGDR represents an area of approximately 0.21 km^2.

The MEGDR topography data is stored in several different files with the .img extension. Each file contains sequential height values for the area it defines and is accompanied by a .lbl file containing information describing the range of latitudes and longitudes it covers. In order to read data from these files, a MEGDRReader class based on the Java NIO API was developed. An instance of MEGDRReader provides methods for retrieving a height map based on line and sample numbers within a particular MEGDR file.

In order to hide the details of the read operations, a MEGDR class was built on top of MEGDRReader. Each MEGDR instance handles one .img file and processes information from the corresponding .lbl file during initialization. Using this information, each instance is able to provide height maps based on longitudes and latitudes rather than on meaningless line or sample numbers within a .img file.

To abstract the data loading process even more, a DataManager class was developed to handle the topography data for an entire data set (e.g. the MEGDR topography data for Mars). A particular DataManger may be configured to manage all input from a number of objects that implement the DataSource interface. DataManager implements the DataProvider interface whose methods specify the ways in which other parts of the system can request height maps or other data.

There is no guarantee all the data to fulfill a particular request for a height map will be contained in one file. DataManager handles this gracefully by creating an empty height map of the correct size and filling it in one piece at a time. Additionally, loading a height map may take a significant amount of time. To prevent this from interfering with other parts of the running application, a request/callback architecture is used. The DataManager receives a DataRequest object through a method that returns immediately. It then makes a callback to an instance of DataConsumer once the height map has been loaded by a separate thread.

NASA has also developed a set of image maps for Mars. The Mars Global Digital Image Mosaic (MDIM) [5] pieces together approximately 4600 images taken by the Viking Orbiter and is available in either grayscale or color. The images are available in several different resolutions and align quite well with the topography data. This allows the images to be used as a natural texturing for the landscape and provides additional realism in some cases.

The architecture for loading texture data from the MDIM is very similar to that for loading topography data from the MEGDRs. The only major difference is that texture data can be stored in two different forms. The original MDIM is available in JPEG and Isis Cube formats. The JPEG images require significantly less storage space due to compression but also take longer for the application to read and process. Either may be used through the ImageTextureSource and BinaryTextureSource classes.

3.2.2 Building Terrain

Once a height map is loaded into the program via the DataManager, it is passed to the constructor of a SphereTerrainSection. The SphereTerrainSection class essentially extends the built-in Java3D Shape3D class and serves as the internal representation of the actual pieces of terrain the application displays. When first constructed, a SphereTerrainSection calculates the coordinates of a basic wireframe mesh for the terrain as seen in figure 2. These coordinates are processed in such a way that the height values are mapped to fit on the planet's spherical surface. In order to emphasize features of the landscape, vertical exaggeration may be used. A vertical exaggeration of approximately 3x is used for all figures shown.

Following construction of the wireframe, the SphereTerrainSection makes use of the Java3D utility classes GeometryInfo and NormalGenerator to generate appropriate surface normals for use in lighting the mesh. Another utility class, Stripifier, is used to convert the rectangular grid of coordinates into a long strip of triangles for more efficient rendering. All these elements are contained in a Java3D Geometry object which is attached to the SphereTerrainSection.

After the geometry of the terrain has been established, an instance of the Java3D Appearance class is used to specify shading and lighting. The Appearance is assigned a Java3D Material object to determine how it is shaded based on values for ambient, diffuse, specular, and emissive colors (see Figure 3). In addition, the Appearance object contains any texture information for the terrain. If texturing is desired, an appropriate texture is loaded from the MDIM and applied to the terrain. Java3D provides several different algorithms for combining the texture with the material applied to the terrain.

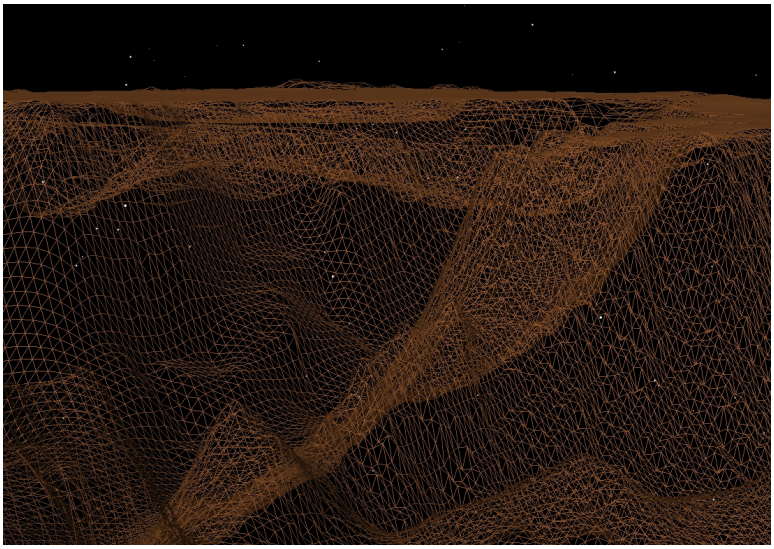

Figure 2: Wireframe terrain.

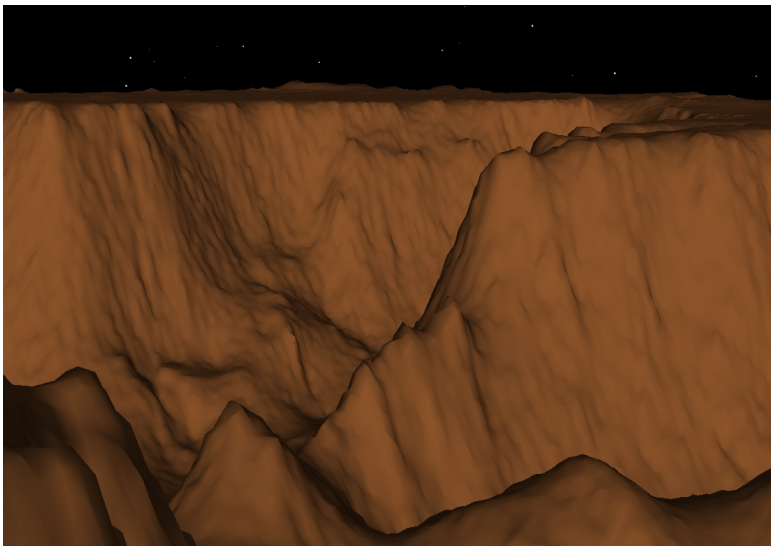

Figure 3: Solid terrain without texturing.

3.2.3 Putting it all together

The Planet class is responsible for organizing the virtual world and for providing the bridge between the content and view subgraphs in Java3D. Planet handles lighting, the background/sky, and several elements of initialization. However, control over the landscape is subordinated to an instance of the LandscapeManager class.

The LandscapeManager keeps track of all SphereTerrainSections for the planet and

handles the creation and removal of these sections from the virtual world. A LandscapeManager is associated with a particular DataProvider and TextureProvider which provide desired topography and texture data, respectively. The LandscapeManager also provides information about the currently loaded landscape through a number of public methods.

3.2.4 Level of Detail (LOD)

Since the MEGDRs contain several gigabytes of pure height values, loading the entire Martian landscape at the highest resolution available is not feasible. Rather, this application has been designed to adjust the level of detail of particular terrain sections at real time. This makes it possible for a user to explore several different areas without restarting the application and manually loading a new area. An instance of LODLandscapeManager, a subclass of LandscapeManager, is used when real time LOD is desired.

The level of detail for a particular terrain section is based on its distance from the viewer's location in the virtual world. The distance at which each terrain resolution is loaded may be set from the user interface. For example, resolutions of 4 pixels per degree can be loaded for the entire planet, but resolutions of 128 pixels per degree might only be feasible within 400,000 meters of the viewer. These ranges vary greatly with the machine on which the application is run and especially with the amount of memory available. Texturing the landscape also increases the memory requirements for a particular area.

Constantly iterating through each possible section that might need to be added or loaded at new level of detail is inefficient, so the LOD algorithm only checks those sections that are likely to change. These sections are determined by starting directly under the viewer with respect to the center of the planet and proceeding outward in a spiral manner. Once a full "loop" of the spiral is completed without detecting any necessary additions in that loop, no other sections will be close enough to merit level of detail changes, and the search finishes. This process is repeated once for each new frame of the visualization.

There are a number of potential issues with this algorithm. If the loading distance is set too high for any of the resolutions, the outward spiral may "wrap around" the planet and enter an infinite loop when it runs into itself. The algorithm has been modified to prevent this and places a hard limit on the number of times the spiral may loop. Another potential problem occurs when the viewer moves to a different location very quickly. Because the number of pending data requests at a given time is limited, not all sections may be removed even though they are not needed. To combat this, the user may toggle whether the LOD algorithm also iterates through each loaded section to ensure it is actually needed.

Additionally, at the user's discretion, the LOD algorithm also considers the direction in which a viewer is looking. The algorithm is able to take into account whether terrain sections should theoretically be visible to the viewer based on the curvature of the planet and the altitude of the viewer. Several sections are excluded by this criterion if the user is near the planet's surface. Additionally, the algorithm contains optional logic to include only those sections contained within the current view frustum. This criterion becomes especially useful when generating a video based on a prerecorded path because the user cannot suddenly turn around to see a blank landscape.

3.2.5 Gluing the landscape together

Another potential problem arises from the use of real time LOD in that terrain sections will not correctly align with their neighbors if the two are not at exactly the same resolution. This leaves the visualization with obvious and disconcerting edges between each section and destroys the appearance of a continuous landscape. To solve this problem, each section is glued to its neighbor by a special mesh that smooths the transition from one resolution to another.

Solving this problem has been one of the most significant challenges in designing this application. The current solution requires each section to keep track of the coordinates along each one of its edges and the surface normals used for lighting at those points. From this information, an instance of the Glue class is created and used to fill the empty space between terrain sections. Each SphereTerrainSection has an associated Glue that handles the terrain on its bottom and right edges. Furthermore, whenever a neighboring section that touches the Glue is changed, the old Glue must be discarded and a new Glue created.

For each new Glue, an appropriate mesh must be created that seamlessly connects each neighboring piece. To do this, the resolutions of all pieces involved must be considered. In order to connect the edge of one SphereTerrainSection to another, there must be an equal number of points on both sides of the Glue. If this were not the case, "tears" would still exist in the landscape. Since all the sections may be at different resolutions, auxiliary points are generated by interpolating between the coordinates on the edge of the sections with smaller resolutions. The coordinates are then used to create a mesh that seamlessly fills the space between neighboring sections. Lighting is also an issue, and the normals corresponding to the Glue mesh are computed by considering the normals on the edges of the neighboring SphereTerrainSections.

3.2.6 Building a Graphical User Interface (GUI)

In order to control the visualization, a GUI was designed using the Java Swing toolkit [3]. As previously mentioned, quick integration with Swing has been a benefit of using

Java3D. The GUI is designed to be as minimal as possible so as not to interfere with the visualization. One particular obstacle overcome during the creation of the GUI was compatibility with a stereoscopic display system. Two identical views of the same interface are needed to avoid problems when the stereo images are projected on top of one another. Also, the two interfaces must both appear to respond to any user interaction. The Interface class handles the interactive pieces of the GUI and provides for the "cloning" of one interface to another.

3.2.7 Navigation

The application developed during this project offers two different methods of navigation. The first method allows the user to investigate different areas of the planet by controlling his or her location and orientation within the virtual world. This "spaceship" navigation may be controlled through the keyboard, by specifying a particular location for fast travel, or by following a prerecorded path. Additionally, a Nintendo Wii Remote may be connected to the application via Bluetooth and serve as a controller. By holding the Wii Remote as if one were holding a "virtual airplane," one is able to "fly" over the surface of the planet in a natural way. The other form of navigation allows the user to look around without changing the direction in which the "spaceship" is moving. This is especially useful for integrating head tracking into the visualization.

3.2.8 Creating Videos

In addition to use as a real time interactive visualization, the planet viewing application may be used to generate videos. In order to generate a video, one must first specify a path along which the visualization will be recorded. To create a Path, one may manually move the camera to several points along the path, and the application will interpolate between these points. Alternately, the user can set the application to record navigation and use any other navigation tools to move along the path desired for the video. This method is especially useful when controlling navigation with the Wii Remote.

The MovieMaker class is used to produce videos based on a particular path. Options that may be set through the user interface determine how a video will be produced. Users have the option to control the dimensions of the video as well as to specify whether or not two images for a stereoscopic video are desired. The speed at which the visualization moves along the prerecorded path can also be specified.

Before generating a video, the user must select a folder into which images for each frame of the video will be saved. Based on the desired starting point and length of the video, the MovieMaker class will assume navigation control and step through the visualization along the current path. After each step, the MovieMaker will stop and wait until all level of detail adjustments to the landscape are complete before capturing the frame and saving

it as an image. If the stereo option has been selected, two images will be saved for each frame to enable the production of stereoscopic videos. One problem with the manner in which images are captured is the mouse cursor or any other window passing in front of the application's display will appear in the image. It is the author's intent to remove this bug in future versions.

Unfortunately, the video production process is not completely integrated into the current application. Each video is exported as a sequence of images for each frame which must be compiled into video format. Several free tools are available for doing this. FFmpeg [6] is one such tool and has been used to compile many videos for this application.

3.2.9 Virtual Reality

Java3D provides excellent built-in support for use in virtual reality environments. One example of this is the transparent generation of stereoscopic views by the framework. Stereoscopic visualization involves the use of two separate images, one for each eye, to create the illusion of depth in an image. In Wartburg College's Stereoscopic Visualization Environment (SVEN) [7], passive stereo has emerged as the most promising technique for large audiences. In Wartburg's SVEN, two projectors are used to project both images through filters such that the images are polarized in a mutually perpendicular fashion. Viewers are equipped with an inexpensive pair of polarized glasses, again with each lens polarized mutually perpendicular to the other. This allows each eye to receive only the correct image and produces the desired effect.

The simplest way to provide one image to each projector is to connect both projectors to the same computer and allow each to handle half the display. An application only needs to show two images side by side with each filling half the display. This is accomplished through the StereoViewer class which does little more than wrap the stereo capabilities already included in Java3D in a convenient GUI component. The StereoViewer contains two separate Canvas3D objects, each tied to the same instance of a Java3D View. The only configuration necessary is to specify the physical separation between a user's eyes and an appropriate field of view.

Java3D also includes built-in support for head tracking. Because the view model provides methods for specifying the relationship between the virtual and physical world, the application only needs to specify a set of transformations from the user's head to Java3D's coexistence coordinate system. Coexistence coordinates are defined to exist in both the physical and virtual worlds and thus allow Java3D to appropriately adjust the view based on the user's head position. While this setup is limited to only one user at a time, it provides a uniquely immersive visualization for that one user.

At Wartburg College, research has been underway to develop a head tracking system using the infrared camera inside the Nintendo Wii Remote [8]. The Wii Remote may be

connected to a computer through Bluetooth and to a Java application through the WiiuseJ library. Once connected, the Wii Remote provides tracking information for two infrared LEDs attached to a pair of glasses. Using this information, the application computes the location of the user's head relative to the Wii Remote. While using only two LEDs slightly limits a user's range of motion, the experience is still realistic and gives the illusion of looking out a window at the planet's landscape.

3.2.10 Algorithmic Terrain Generation

Even though the MOLA topography data is only available in resolutions up to 128 pixels per degree, it is possible to algorithmically create sections of terrain with higher levels of detail. Algorithmic terrain generation is handled by the TerrainGenerator class and based on an algorithm described by Paul Bourke [9]. If the generated terrain is based on the highest resolution data available for the actual landscape, the "fake" terrain preserves all the major features of the actual terrain while increasing the apparent level of detail. Generated terrain also adds the appearance of "roughness" to the terrain which may or may not be desirable for particular visualizations. Additionally, generated terrain both takes longer to load into the visualization and the higher resolutions for generated terrain lead to greater memory requirements for the same terrain.

4 Conclusions

The planetary visualization tool developed during this project has shown the Java3D framework is capable of supporting an interactive and immersive visualization on a large-scale. While it does not have the raw power of a low-level framework, Java3D provides enough rendering efficiency to display an extensive landscape. Using Java3D's compiled-retained rendering mode especially contributes to rendering efficiency. In addition, the framework's is able to support real time level of detail changes to the landscape.

In addition, Java3D's view model has provided an excellent starting point for integrating visualizations with virtual reality environments. Java3D's built-in support for stereoscopic viewing has allowed easy integration with Wartburg's Stereoscopic Visualization Environment (SVEN) [7] and other similarly designed systems. Java3D's view model also makes head tracking straightforward. At Wartburg, a head tracking system based on the Nintendo Wii Remote has been tested with the planetary visualization tool.

Developing the application itself has also been a success. The final application supports both interactive and prerecorded visualizations both on a normal computer and in Wartburg's SVEN. The application supports real time level of detail adjustments to the landscape and is able to use images from the MDIM to texture the landscape. Several

different methods of navigation have been implemented, and several configuration options are available from a GUI. The visualizations have also received positive feedback from several viewers.

5 Future Work

The visualizations currently lack a realistic sky to accompany the Martian landscape. The stars currently shown in the background could be adjusted to represent the real distribution of stars as viewed from Mars, and other objects could be added to the night sky to represent planets, Mars's two moons, or even satellites currently orbiting the planet. A daytime sky could also be implemented and lighting could be determined by the actual position of a virtual sun.

The educational value of the application could be augmented by introducing information about the particular landscape features on Mars. For example, a list of interesting areas could be compiled and displayed in the applications GUI. A user would then choose an area to visit and perhaps fly along a prerecorded tour around the landscape. Information could be displayed as text during the tour or spoken aloud by a text to speech engine.

NASA provides many resources in addition to the MEGDR and MDIM datasets. In particular, very high resolution topography data is available for several small areas on the planet. These areas could be introduced into the application as small "areas of interest" within the context of the larger landscape. Highly detailed terrain in these sections would also give users a better sense of the magnitude of the rest of the Martian landscape.

General improvements to the visualization tools are also planned. The application has yet to be thoroughly tuned for performance. Minimizing the number of polygons used to model relatively flat terrain could significantly increase both rendering speed and the detail to which the landscape may be loaded. The applications code also needs to be reorganized and cleaned up in some areas, and it is the author's intention to eventually make the source code for the application openly available.

6 Acknowledgments

Work conducted to integrate interaction with the Nintendo Wii Remote with this application was partially supported by a Maytag Innovation Award for student/faculty research. I would like to personally thank Dr. John Zelle, my advisor for this project. I thank NASA, the developers of Java3D, and the open source community for making this project possible. Finally, I would like to thank anyone who has taken the time to be interested in this project or offered encouragement even as simple as "Hey, that's neat."

References

[1] "PDS Geosciences Node Data and Services: MGS MOLA MEGDRs," Apr. 19, 2007. [Online]. Available: http://pds-geosciences.wustl.edu/missions/mgs/megdr.html.

[2] "The Java 3D API Specification," Jun. 2002. [Online]. Available: http://java.sun.com/javase/technologies/desktop/java3d/forDevelopers/J3D_1_3_API/j3dguide/index.html.

[3] "Trail: Creating a GUI with JFC/Swing (The Java™ Tutorials)," Feb. 14, 2008. [Online]. Available: http://java.sun.com/docs/books/tutorial/uiswing/.

[4] "Lesson: Concurrency (The Java™ Tutorials > Essential Classes)," Feb. 14, 2008. [Online].Available:http://java.sun.com/docs/books/tutorial/essential/concurrency/.

[5] "USGS Astrogeology: Mars MDIM Digital Image Mosaic," Sept. 4, 2008. [Online]. Available: http://astrogeology.usgs.gov/Projects/MDIM21/.

[6] "FFmpeg," Mar. 10, 2009. [Online]. Available: http://www.ffmpeg.org/.

[7] J. M. Zelle and C. Figura, "Simple Low-Cost Stereographics: VR for everyone. *SIGCSE Bulletin*, 36(1)(348357), March 2004.

[8] S. Herzberg, "Wiimote Interaction for Virtual Reality Applications," *in Proceedings of the Midwest Instruction and Computing Symposium.* April 2009.

[9] P. Bourke, "Fractal Landscapes," January 1991. [Online]. Available: http://local.wasp.uwa.edu.au/~pbourke/fractals/noise/.