WestVirginiaUniversity
THE RESEARCH REPOSITORY @ WVU

Graduate Theses, Dissertations, and Problem Reports

1999

Mission planning and remote operated vehicle simulation in a virtual reality interface

Christopher Samuel Allport
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation

Allport, Christopher Samuel, "Mission planning and remote operated vehicle simulation in a virtual reality interface" (1999). *Graduate Theses, Dissertations, and Problem Reports*. 940.
https://researchrepository.wvu.edu/etd/940

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

Mission Planning and Remote Operated Vehicle Simulation in a Virtual Reality Interface

by

Christopher Samuel Allport

B.S.E.E, West Virginia University

THESIS

Submitted to

the Department of Computer Science and Electrical Engineering of

the College of Engineering and Mineral Resources

at

WEST VIRGINIA UNIVERSITY

in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

Committee

Dr. Biswajit Das

Dr. Wils Cooley

Dr. Mark Jerabek

Morgantown

West Virginia

1999

Copyright 1999 Christopher S. Allport

ABSTRACT

Mission Planning and Remote Operated Vehicle Simulation in a Virtual Reality Interface

Christopher S. Allport

Virtual reality simulations are finding applications in a wide range of disciplines such as surgical simulation, electronics training, and crime scene investigation. During the Mars Pathfinder Mission, in summer 1997, NASA scientists unveiled a new application of virtual reality for the visualization of a planetary surface. The success of this application led to a more concentrated effort for using virtual reality visualization tools during future missions. The thrust of this effort was to develop a new interface which would allow scientists to interactively plan experiments to be performed by the mission robots. This thesis covers two of the primary aspects of implementing this system. The first topic was to develop a kinematic model for one of NASA's rovers for use in a virtual reality simulation. The second aspect of this thesis is the implementation of the tools required for the mission planning module, which are the interfaces that the scientists use to plan the experiments for the rover.

Acknowledgments

I wish to acknowledge my advisor, Dr. B. Das, for his encouragement in pursuing this research. I would also like to thank the other members of my committee, Dr. W. Cooley and Dr. M. Jerabek. I am grateful for their guidance and valuable suggestions in writing this thesis. The Intelligent Mechanisms Group at NASA-Ames Research Center in Mountain View, California, was instrumental in introducing and getting me involved with this research. Special thanks to Dr. Theodore Blackmon who worked to get me involved and keep me involved with this project. I also wish to thank Erik Shreve and Paul Sines for their help. Special thanks to my family for their encouragement and support throughout my education.

Contents

Abstract ii

Acknowledgments iii

List of Figures vii

1 Introduction 1

2 Background 4

2.1 Overview of Virtual Reality 4

2.2 Goals of Research 5

3 Implementation of Terrain Following 6

3.1 Methods of Terrain Following 6

3.2 Rover Kinematics 9

3.3 Rover Drive Interface 14

3.4 Improvement Over the Original System 16

4 Implementation of Mission Planning Module 17

4.1 Modularity of Design 17

4.2 Elements of the Mission Control Module 18

4.2.1 General Properties User Interface 18

4.2.2 Panorama Image Sequence Visualizer 19

4.2.3 Navigation Camera Visualizer 21
4.2.4 Spectrometer Visualizer 22
4.2.5 Arm Task Planner 25

5 Results & Conclusions 26
5.1 Accomplishment of Project Goals 26
5.2 Validation at the Field Test 27
5.3 Modularity 28
5.4 Collision Detection 28
5.5 Discussions 28

6 Future Work & Suggestions 31
6.1 Collision Detection 31
6.1 Rover and Terrain, Objects, and Other Rovers 32
6.2 Improvements in Collision Detection 35
6.2 Improvements in Terrain Following 37

Bibliography 38

Appendix

A Explanation of Virtual Reality Concepts 40
A.1 The Scene Graph 40
A.2 Reference Frames 42
A.3 Progressive Transforms 44

B Pan-Tilt Sequence Planner Source Code 46
B.1 Listing of panCam.h 46
B.2 Listing of panCam.c 46

C Spectrometer Planner Source Code **74**

C.1 Listing of spectral.h 74

C.2 Listing of spectral.c 74

D Navigation Camera Planner Source Code **98**

D.1 Listing of navCam.h 98

D.2 Listing of navCam.c 98

E Terrain Following Source Code **112**

E.1 Listing of marsokhod.h 112

E.2 Listing of marsokhod_sim.h 115

E.3 Listing of marsokhod_sim.c 115

List of Figures

1-1 Top View of MarsMap 2
3.1 "Move and Settle" Terrain Following 7
3.2 Surface Determination and Occlusion Problem 8
3.3 "Drop in Place" Terrain Following Method 9
3.4 Degrees of Freedom of Marsokhod Rover 10
3.5 Calculating Roll of the Marsokhod Rover 11
3.6 Calculating Pitch of the Marsokhod Rover 12
3.7 Illustration of Small Angle Assumption 12
3.8 Marsokhod Drive Panel 14
4.1 General Task User Interface 18
4.2 Panoramic Image Tool Control Panel 19
4.3 Panorama Image Sequence Tool 20
4.4 Navigation Camera Control Panel 21
4.5 Navigation Camera Imaging Sequence 22
4.6 Spectrometer Visualization Tool Shown Capturing a Rock 23
4.7 Spectrometer Visualization Tool Shown Bending around the Corner of a Box 24
4.8 Spectrometer Visualization Tool Showing Capture Area with an Obstruction 24
4.9 Arm Task Planning Panel 25
A-1 Sample Scene Graph 41
A-2 Separator Node 42
A-3 Illustration of object moving along local and world axes 44
A-4 Progressive Transform 45

Chapter 1

Introduction

“I have to see it to believe it.” These words encapsulate many individuals’ thoughts about progressive ideas. However, it is not always feasible to actually show someone the idea or even a physical model of it. On the other hand, it may be possible to create some illustrations or presentation which would better express the idea. Visualization is an important step for most to understand complex concepts, and traditional computer technology has played an important role in visualization in the form of two dimensional graphics, animations, and sound. Virtual reality offers a much better solution to using computers as visualization tools. It creates an interactive, three dimensional simulation which can be extremely realistic in its appearance and level of interaction; thus, it can play an important role in intelligent data gathering. Virtual reality is finding applications in disciplines ranging from surgical simulations to automobile design.

A recent application of virtual reality is space exploration as demonstrated during the Mars Pathfinder mission. The data received from the Pathfinder landing module and exploring rover were used in a virtual reality application to create a three dimensional reconstruction of the Martian topography. This planetary visualization tool, developed for use at mission control during the Mars Pathfinder mission, was named MarsMap. It enabled scientists to obtain data about the Martian surface in a completely new way [1]; they were able to measure and view the surface of Mars in three dimensions. Figure 1.1 shows an overhead view of the Pathfinder landing site as visualized using MarsMap.

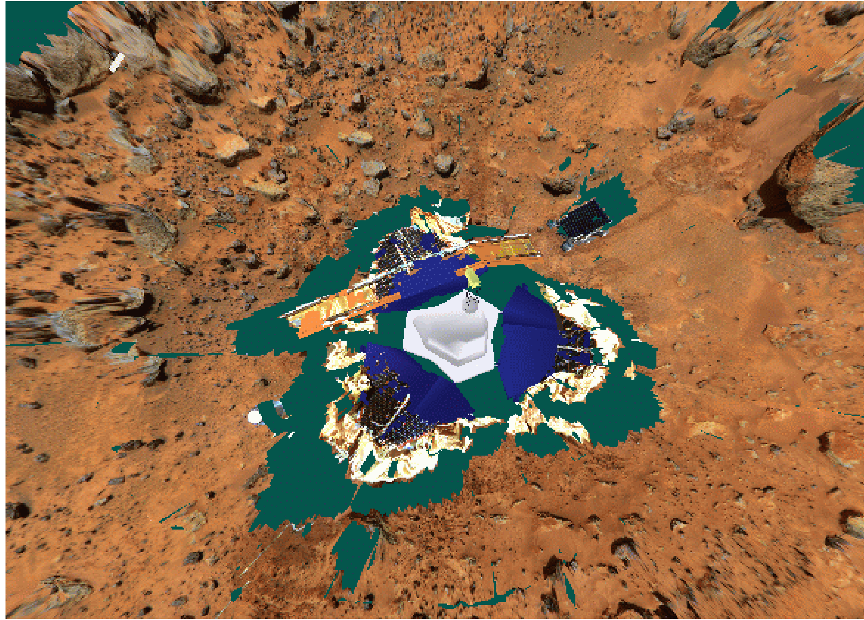

Figure 1.1 *MarsMap view of Pathfinder Landing Site*

MarsMap was a revolutionary system, but fell short in many ways as a comprehensive mission visualization tool. It allowed the scientists to make measurements and view the terrain in three dimensions, but it did not provide any mission planning tools nor did it provide any rover visualization. Based on these considerations, and the experience gained from the Pathfinder mission, it was concluded that an intuitive and modular user interface that will allow geologists to plan experiments in a virtual environment before performing the experiment with an actual robot will be of significant value for future missions. There are additional concerns, however. Such an interface would have to be flexible. Each mission has a different robot with a different set of tools on board. How then, can such an interface be constructed so its use will transcend a specific mission and be applicable to any mission?

The excitement inspired by MarsMap led NASA scientists to start working to expand upon its interface [3] and add a rover simulation after the Pathfinder mission concluded. With the new additions, it was possible to realistically drive the mission rover, Sojourner, on the Martian surface in real-time. It also allowed the scientists to replay different parts of the mission by placing rover

models at locations of mission events. MarsMap was even used to compare the actual and intended paths traversed by the rover. The rover could also be placed in different locations, and billboards detailing the images captured by the rover at those points could be displayed [4, 5]. However, at mission control, scientists could not drive the virtual Sojourner model over the Mars terrain nor did any of these additional features exist. These additions could have provided a better method for optimized data gathering and an improved mission outcome.

The objective of this thesis was to address the shortcomings and improve upon the MarsMap system. The original system had problems with its terrain following algorithms. Therefore, the first task consisted of developing terrain following algorithms for the Marsokhod rover, the NASA-Ames Intelligent Mechanisms Group's test-bed rover.

Additionally, the original interface only had mensuration tools allowing the scientists to gather data about the surface. There existed no means to interact with the environment for the purpose of planning. Thus, after the first phase, work shifted to developing a modular mission control interface. This work was responsible for the development of many of the tools and user interfaces used in the mission control module.

Finally, this thesis served to develop ideas for collision detection. While these ideas were not implemented, they do serve as a starting point for integrating collision detection into the visualization system that will be important for future missions.

Chapter 2

Background

This chapter provides a brief overview of the virtual reality technology and describes the objectives of this research. For a more detailed description of virtual reality, please refer to Appendix A of this thesis.

2.1 Overview of Virtual Reality

The virtual reality (VR) technology was first proposed and demonstrated in the 1960's, where individual display units were mounted to a common pole which allowed users to change the position of their viewpoint of a wireframe cube [2]. As computer processing power increased, the complexity of virtual reality systems also increased. Modern VR systems have moved from simple wireframe models to thousands of complex, fully textured, shaded, and anti-aliased polygons. While virtual reality still requires a significant amount of processing power, it has started to reach the consumer market by way of video games. Although full scale virtual reality has yet to hit mainstream technology, its value as a visualization tool, as a form of entertainment, and as an unlimited source for exploration are easily recognized.

Virtual reality uses a combination of computer graphics, animation, hypertext, video, and sound to create a highly interactive environment in which a person can interact with virtual objects [6, 13]. An excellent description of virtual reality is described by Howard Rheingold in his book, *Virtual Reality* [7]. Rheingold says:

> "Imagine a wraparound television with three-dimensional programs, including three-dimensional sound, and solid objects that you can pick up and manipulate, even feel with your fingers and hands. Imagine immersing yourself in an artificial world and actively exploring it, rather than peering in at it from a fixed perspective through a flat screen. Imagine that you are the creator as well as the consumer of your artificial experience."

Rheingold enthusiastically describes virtual reality with somewhat fantastic stereotypes, but he makes his point. Virtual reality is a multi-sensory, immersive computer interface bound only by the imaginations of the creators.

Although this technology has been around for several years, it is still in its infancy because computers have not been powerful enough to render virtual reality simulations. Now that computers are significantly more powerful, it is conceivable to apply virtual reality to almost any discipline, thus leaving a great deal of research to be undertaken.

2.2 Goals of Research

This thesis addresses two primary areas:

1. articulate the Marsokhod rover model so it will, with reasonable accuracy, traverse over the model of the terrain at a mission site; and
2. assist in development of a modular mission goal planning interface which will allow geological scientists to interactively plan mission experiments and obtain preliminary visualization information about these experiments before they are ever performed.

In addition, this thesis also reports on the development of strategies for collision detection which could ultimately be integrated to the interface for more realistic rover operation.

Chapter 3

Implementation of Terrain Following

Terrain following is an important requirement for robotic exploration of a planetary surface; it is desired that as the rover drives over the terrain it maintains a consistent height above the terrain. For single or multiple independent points, this is a relatively trivial problem. However, when multiple points of one particular object are following the terrain, such as the wheels of a rover, the problem becomes quite complicated. To solve this problem, a clear understanding of the kinematics of the terrain following object is necessary and simplifying assumptions need to be made. In this part of the research, an algorithm for terrain following for the Marsokhod rover was developed and implemented in virtual reality.

3.1 Methods of Terrain Following

The first goal of this thesis was to incorporate a fully articulated (each individual piece moves) model of the Marsokhod rover. In this simulation, the rover was to be able to drive over the surface of the Martian data set, pitching and rolling as the physical rover would. Basically, each of Marsokhod's six wheels needs to be situated onto the data set. After the wheels are in place, it is critical to know if the rest of the rover can actually move to where it was trying to go [this is a separate problem which is handled independently].

The first step was to decide exactly how, algorithmically, the position of the wheels would be calculated. The approach initially considered can be best described as "move and settle." Using

this method, the rover geometry is moved into its new position and then is settled onto the data set. Figure 3.1 schematically shows the steps involved in the "move and settle" method. Figure 3.1(a) shows the rover in its initial state. As described above, the next step is to translate the rover forward, shown in Figure 3.1(b). Finally, in Figure 3.1(c), the front, middle, and rear axles are translated (vertically) and rotated to fit the terrain.

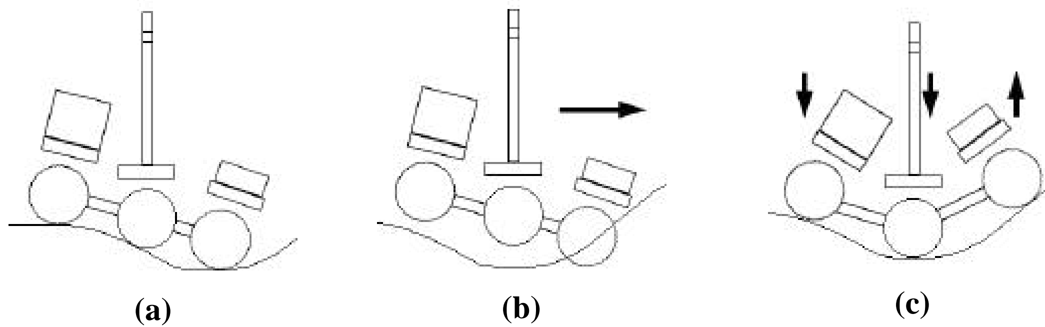

Figure 3.1 *"Move and Settle" Terrain Following*

The primary advantage of the "move and settle" method is that it provides an incremental position adjustment from frame to frame and ensures that the wheels will settle onto the proper surface. This algorithm is difficult to implement, however. At each new position, it requires looking for the terrain above *and* below the axis of the wheel and must make a determination which, if either, is the correct surface. For example, for the front axle in Figure 3.2, there is a surface above the axis center as well as a surface below the axis center. While it may be perfectly clear to a human on which surface the wheel should be placed, it is not obvious to the computer. After finding the correct surface, the software must then determine if the wheel will fit into the new position. This figure also illustrates how a surface feature, like a rock, may keep the wheel from fitting into place.

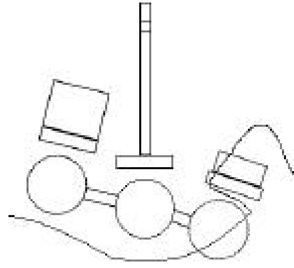

Figure 3.2 *Surface Determination and Occlusion Problem*

While this method has some other benefits as discussed later [see Collision Detection], it was abandoned for a slightly less robust, hence simpler to implement, method. The technique used to implement terrain following for the Marsokhod rover can be classified as “drop in place.”

Preliminary testing indicated that the “drop in place” method was the best immediate solution for articulating the Marsokhod rover. In this method, the rover is lifted above the surface (NOTE: it must be lifted a distance greater than or equal to the height of the highest local feature), the pan and tilt of each axis is calculated from the distance of the center of the wheel to the surface, and then the rover is translated down into the proper position. The front and rear axles were attached to the middle axle thus allowing them to move with the middle axle.

The following figures illustrate how the “drop in place” method works. Figure 3.3(a) shows the rover in an initial settled position. In Figure 3.3(b), the rover is translated up and the angles are reset. Figure 3.3(c) shows the axle angles as set for the new terrain, and Figure 3.3(d), the rover is lowered into its new position.

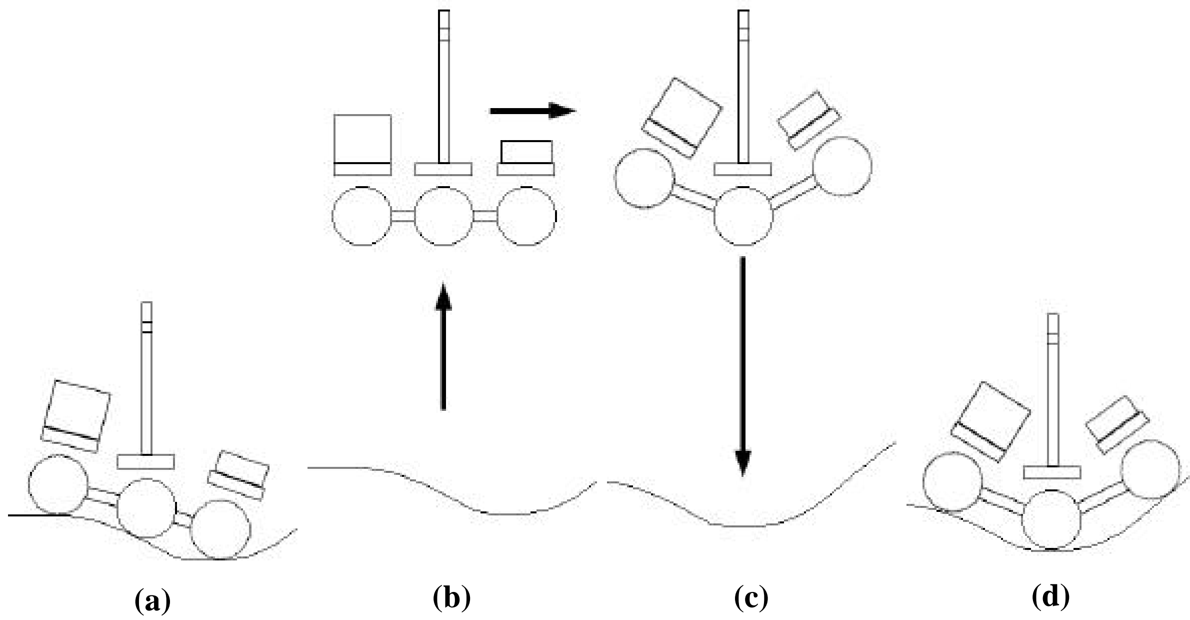

Figure 3.3 *"Drop in Place" Terrain Following Method*

Although terrain following was the first goal, before it could be addressed a subroutine had to be developed to load and build the articulated model of the Marsokhod rover. Work then continued to implement the actual terrain following.

3.2 Rover Kinematics

To simulate the physical interaction of the rover with the terrain, the virtual rover had to be programmed with its appropriate kinematic properties. Figure 3.4 shows a schematic of the axes of rotation of the rover. The front and rear axles can pitch about the center axle, and the front and rear axles can roll about the shaft connecting them to the center axle.

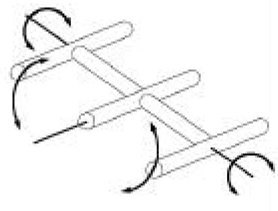

Figure 3.4 *Degrees of Freedom of Marsokhod Rover*

Several assumptions were made in the development of the kinematic equations to simplify and speed up the calculations. The first assumption is that the rover can go to any location where it is placed. This is, of course, an inaccurate premise as it is clear that a rover would be hard pressed to climb, for example, an 80% grade. These problems, however, can be circumvented by programming the physical limitations of the rover, but was not attempted for the sake of simplicity.

Since the scene graph was built so the front and rear axles hinge on the middle axle, the middle was taken to be the base axle. In the first simplifying assumption, the middle axle was initially considered to be rigid, thus effectively never having any pitch. In reality, the middle axle is cantilevered between the front and rear axles for stability, and its pitch does change. Since a variety of scientific instruments are mounted on the mast, and since the virtual reality is actually being used for experiment planning, this assumption would have introduced significant errors in the simulation. Therefore, it was corrected so that the pitch of the mast (middle axle) behaves in a manner representative of the actual rover (the calculations will be covered below).

Given that the middle axle was the best one to choose as the primary axle, the other two axles were added as children of the middle axle in the scene graph. The first step in the process was to find the distance of each wheel of the middle axle to the terrain. This was done by projecting a vertical line straight down from the center of each of the wheels until it intersected with the terrain. This function returned the distance of the axle (at the middle of each wheel on each side of the axle) to the terrain. The height of the middle axle, H_M can be calculated as shown:

$$H_M = \frac{H_{LW} + H_{RW}}{2}$$

where H_{LW} and H_{RW} are the distances of the left and right wheels from the terrain, respectively. The next step was to roll (not illustrated in Figure 3.3) the axle so that the two wheels appeared to be sitting on the terrain. The roll angle, θ, can be calculated by:

$$\theta = \sin^{-1} \frac{H_{LW} - H_{RW}}{W_R}$$

where W_R is a predefined constant representing the distance between the left and the right wheels. Figure 3.5 corresponds to this calculation.

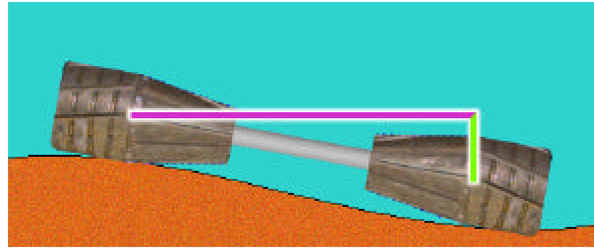

Figure 3.5 *Calculating Roll of the Marsokhod Rover*

The roll of the front axle is then calculated. If H_F is the distance of the front axle from the terrain. The pitch of the front axle can be determined by:

$$\phi_F = \sin^{-1} \frac{H_M - H_F}{W_T}$$

where W_T is predefined as the distance between the front and middle axles. The rear axle was modeled in the same fashion as the front axle, by substituting Φ_R and H_R for Φ_F and H_F. Figure 3.6 illustrates this calculation.

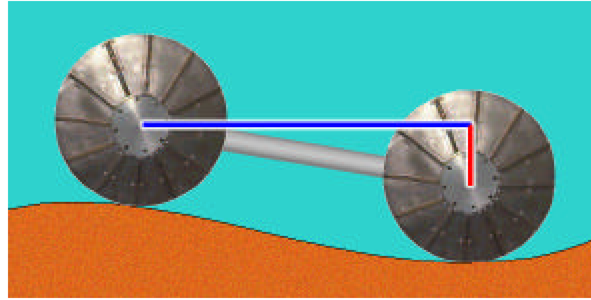

Figure 3.6 *Calculating Pitch of the Marsokhod Rover*

The difference between Φ_R and Φ_F when divided by two, gives the angle through which the middle axle must be rotated to have the mast in the proper orientation.

After all the calculations are completed, the rover is translated down to the surface. It is important to note that H_M is not the distance the axle will be translated down because of the wheel; subtracting the radius of the wheel from H_M results in the proper translation distance.

In addition to the rigid mast, some additional assumptions were made to simplify the kinematics. Although front and rear axles rotate about the middle axle in an arc, the distance used in calculations was for when the front/rear axles were in their initial states. It is possible, by this assumption, that once the axle is rotated, the topology under the wheel will be different and the wheel will not seat properly. Figure 3.7 illustrates this problem. A small amount of accuracy is lost due to this assumption, however, since most of the angles that the rover will be facing are small angles, the error introduced is expected to be quite small.

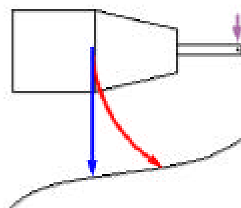

Figure 3.7 *Illustration of Small Angle Assumption - the blue line represents the actual distance*

measured, the red line represents the path along which the wheel rotates, and the violet arrow indicates the pivot point.

Another assumption made introduces errors greater than those resulting from the previous assumption. To significantly simplify the placement of the wheel on the terrain, only a single point from the center of each wheel follows the terrain. In the simulation, it is not uncommon to see the wheel roll through a small rock, or other surface feature. Since the outer half of the wheel is not being tested, it is possible for the edge of the wheel to drive straight through a rock! However, this assumption allows a significant reduction in the amount of programming and in processor load. Also, in most situations, users will not be concentrating on the wheels, and these errors will go unnoticed.

There is another error introduced by the small angle assumption. When an axle drives onto an extremely steep slope, an oscillatory condition arises. In the first measurement, the wheels are found to be further away from the ground than they actually are. This miscalculation is propagated into the next measurement, which then detects the wheels to be too close to the ground. A situation can occasionally be found where the oscillation is damped out after a few cycles. Unfortunately, it is just as possible to find a situation where the oscillation increases. Left alone, this is a very unstable system.

Although the above assumption does not introduce any substantial functional problems (in reality the actual rover should never be driven on such a steep slope and these kinematics are exclusive to this rover), it needed to be addressed for the virtual system to be physically representative of the actual system. Two approaches were considered to allay this problem: (a) rewrite the terrain following system, or (b) implement a damping function for the rover. The first option was eliminated due to time constraints. The idea of the damping function is similar to using a capacitor to filter out unwanted oscillations in electronic circuits. A damping function was introduced in the simulation to compensate for the oscillatory error. If the change in height from one calculation to the next is greater than some constant, instead of being set to that height, the

axle is eased to that height by a small differential amount, delta. This keeps the robot from jumping from one height value to another, since it is changing by a small amount each time. In the worst case, it should ultimately reach a point where it oscillates only by one delta.

As in electrical systems, the use of damping slows down response of the system. If the virtual rover is driven over the terrain at top speed, it will, at times, appear to be immersed in (or floating above) the land. This is due to the fact that the damping function has not caught up to the actual position of the rover. Such situations will occur only in extreme cases and will not affect normal operation of the rover.

3.3 Rover Drive Interface

In order to navigate the rover over the surface of the terrain, a drive interface was implemented. Of primary importance was to create a natural and intuitive interface which would allow a user to drive the rover as it would physically move. Unlike Sojourner, Marsokhod is able to drive and turn simultaneously. The drive panel, therefore, should reflect this feature. Figure 3.8 shows a screen capture of the Marsokhod drive panel. This panel is displayed as an additional window laying atop the main visualization window.

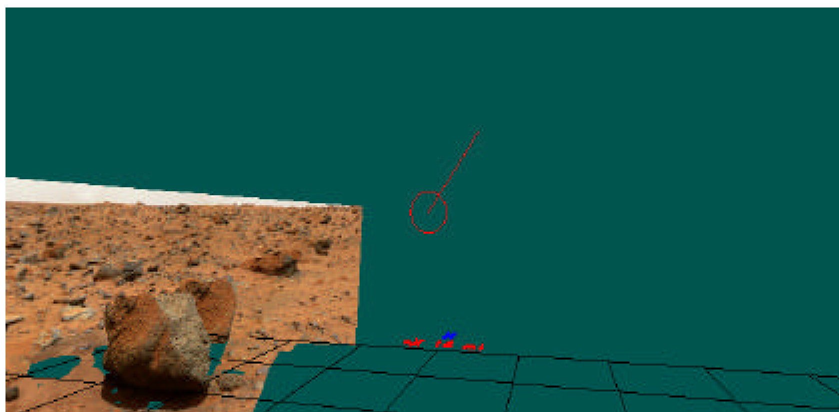

Figure 3.8 *Marsokhod Drive Panel*

The screen is setup as a Cartesian coordinate system with the center of the screen being the origin. The Y-axis is the vertical axis, and the X-axis is the horizontal axis. Any time the left mouse button is depressed in the drive window, the rover will move with a velocity dictated by the equations:

$$V_{drive} = Y_{cursor} * V_{yscale}$$

$$V_{turn} = X_{cursor} * V_{xscale}$$

where V_{yscale} and V_{xscale} are scale factors which adjust the screen coordinates to values respective of the rover speed. The local coordinate frame allows for simple positioning of the rover. The difficulty arises in translating the mouse position on the screen into drive commands for the rover.

To enhance the realism of the simulation, the wheels were rotated as a function of the drive and turn rates. The wheels turn at the forward rate divided by the wheel radius. Since it is capable of turning at the same time, the turn rate divided by the wheel radius is added to the forward rate. This feature makes the rover appear to be driving across the terrain, instead of being scooted across.

A new feature added to the drive panel is the camera mode. Using the right mouse button, the user is able to move the rover's camera around in all directions without moving the rover. The drive window holds the camera in whatever orientation the user sets. This is particularly useful if the user is planning to look in a direction other than straight ahead while driving the rover. This feature allows the user to see, in advance, what view will be given from the rover's camera. The position of the camera can be reset by clicking the center button.

3.4 Improvement Over the Original System

A significant problem that plagued the original system was holes in the data set. In the original terrain following algorithms, the function which determined the heights of each wheel failed each time a hole was encountered. The rover settle function would then try to pose the rover with incomplete information, often causing a snowballing of errors until the program either crashed or slowed the system to a near halt due to processor overhead. This problem needed to be addressed in an effort to make the system more reliable. It was found that this problem could be solved by keeping the previous position of the wheel in memory. If the new wheel position was over a hole, it maintained its previous height. As soon as the center point moves over terrain, it takes on its new height. This enhancement allowed the rover to be driven throughout the entire simulation, irrespective of the presence or the absence of terrain. As an added bonus, this improvement also allows the rover to be driven back onto the landing module, so scientists can more accurately simulate the entire mission — from driving the rover off the lander, to its last known position.

Chapter 4

Implementation of Mission Planning Module

Although virtual reality had been demonstrated by NASA to be useful in a mission control environment (Mars Pathfinder, 1997), it had yet to be used as a planning interface. This chapter will discuss the user interface which allows scientists to plan experiments for the rover. This interface was developed for and tested during the Intelligent Mechanisms Group's 1998-1999 Field Test.

4.1 Modularity of Design

Since NASA typically uses different robots for different missions, it was clear from the onset of the research that the interface would have to be designed in such a way that it was modular and independent of the rover for which the experiments were being planned. The actual development, however, did not result in a fully modular system. While the system had some components of modularity, it was still hard-coded with the simulation of the Marsokhod rover. As time passed, the major planning tools were programmed into the system.

4.2 Elements of the Mission Control Module

The mission control module was comprised of several parts. Of primary interest are the general task planning interface and four of the instrument planning interfaces. Each instrument planning interface has a general task planning user interface associated with it. All of the planning user interfaces are dialog boxes (windows) that lay on top of the main application. As the parameters in the panel are adjusted, the visualization system is updated.

4.2.1 General Properties User Interface

Since the rover can traverse across the data set to constantly allow the user to visualize and plan in multiple locations, it is imperative that the rover be always mobile. To make the planning visualization more clear, a copy of the rover model had to be added, or instanced, at each test site. After a rover is instanced, any of the experiments can be performed. All experiments share a common set of parameters. Therefore, a general user interface was developed for all experiments. A screen capture of the General Task user interface is shown in Figure 4.1.

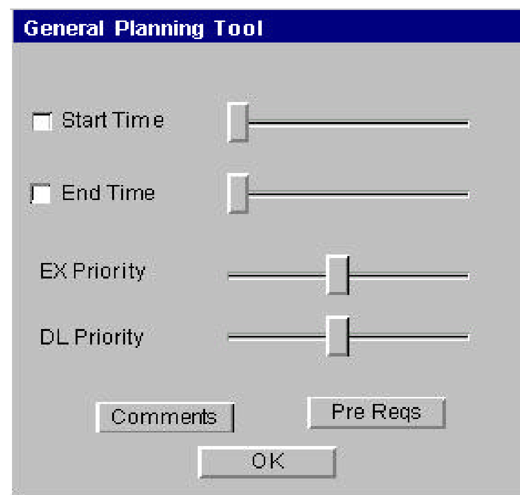

Figure 4.1 *General Task User Interface*

This general task user interface enables the scientists to set specific experimental parameters which apply to all of the experiments. If applicable, a start and/or end time can be specified. These are times for a task to either begin or to finish. The execution priority specifies the importance of a particular task to be executed. The downlink priority supplies information to the rover communication software regarding the importance of the information obtained by this task. If it has a low downlink priority, and a large demand for bandwidth, the data from the particular task may be stored to be sent during a later communication cycle. The comments button opens a text editor (selected by the user) to enter comments about the given task. The prerequisites button, when implemented, will enable the scientists to state what tasks must be accomplished before the current task can be performed.

4.2.2 Panorama Image Sequence Visualizer

In order to help the scientists plan imaging sequences, a tool was developed for doing so in an optimum fashion. Without this tool, it is very easy to overestimate the number of images that needed to be taken to capture the desired region. The control panel, seen in Figure 4.2, allows the user to set the pan and tilt extents, the pan and tilt resolutions, and some various imaging parameters.

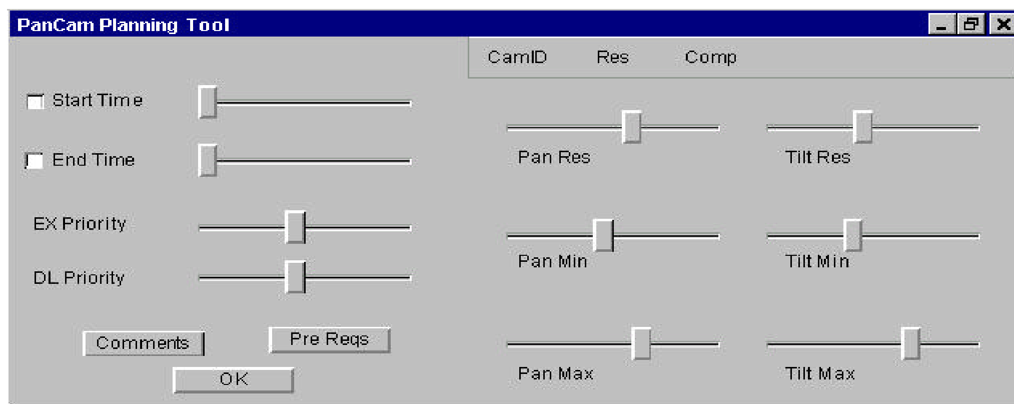

Figure 4.2 *Panoramic Image Tool Control Panel*

This control panel is interfaced with the pan-cam tool. As the pan and tilt extents are adjusted, the 3D representation of the imaging sequence is updated. Figure 4.3 shows the rover in the middle of planning an image sequence.

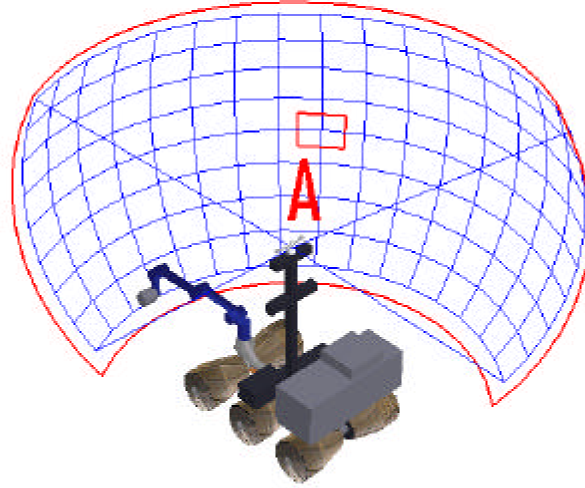

Figure 4.3 *Panorama Image Sequence Tool*

In Figure 4.3, each of the blue vertices represents where an image will be taken. The red line which outlines the blue surface patch represents the total imaging area. This facilitates optimization since the blue vertices mark only the center of the images to be taken. The small red square located in the center of the blue surface patch represents the size of one single image. This gives the scientists an idea of how much overlap and coverage will exist between the images. In order to generate this surface patch, each vertex had to be calculated. This was done at each of the desired pan and tilt stops. If the pan and tilt angles are designated as θ and Φ, respectively. In the world frame, the points are calculated as follows:

$$x = r\sin\theta\cos\Phi$$

$$y = r\sin\Phi$$

$$z = r\cos\theta\cos\Phi$$

where r is the radius of the surface patch to be generated. After generating this geometry in the world frame, it was transformed to the rover's position.

4.2.3 Navigation Camera Visualizer

This tool was developed to help the scientists plan nav-cam imaging sequences and to see the rover's view. Figure 4.4 shows the interface for the nav-cam tool.

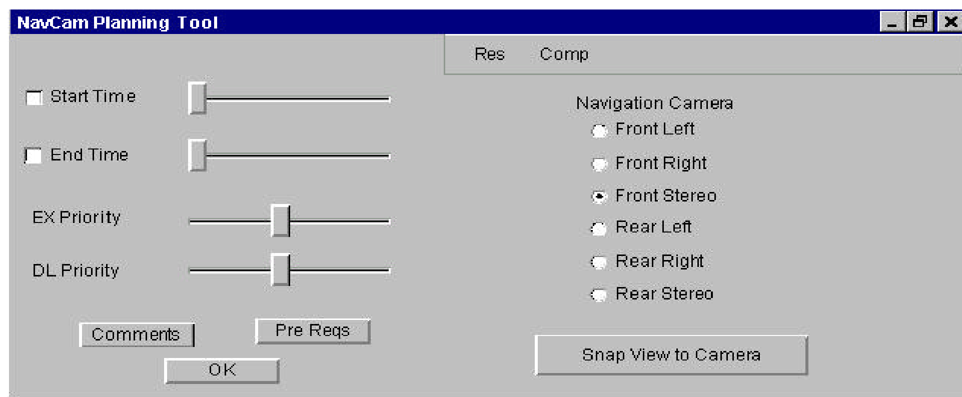

Figure 4.4 *Navigation Camera Control Panel*

As shown in the figure, the user can select between the various cameras, and by selecting "Snap to Camera Position," can view the world from the rover's viewpoint using the specified camera.

Figure 4.5(a) shows what the rover looks like while planning a navigation camera imaging sequence. The red and green projections represent the field of view for the two front cameras. The view from the these cameras can be seen in Figure 4.5(b). The green and red lines are the outlines of the fields of view of the cameras.

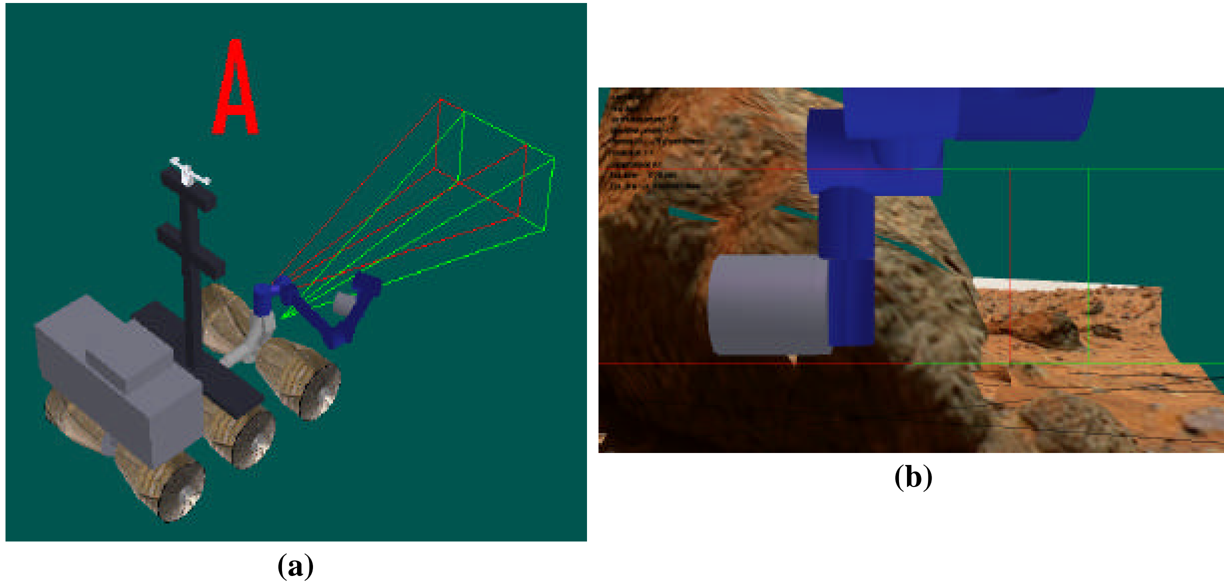

Figure 4.5 *Navigation Camera Imaging Sequence*

4.2.4 Spectrometer Visualizer

The rover also has a spectrometer mounted to it for gathering composition data about geological formations. The spectrometer has a field of view of 1°. This implies that the size of the sampled area is proportional to the distance of the surface from the spectrometer. The area captured by the spectrometer is similar to the surface illuminated by a flashlight. Wherever the light is shown, a circle of light (which grows with distance from the source) appears. If the light is incident on a feature such as a corner, or a step, the light appears on the first incident surface in its path. The spectrometer operates in a similar manner.

Typical use of the spectrometer will be limited to capturing one small coherent surface, but the imaging surface may not completely fill the field of view. Also, if the surface is not even, as in the case of most rocks, it would be useful to see what is being imaged.

This tool was developed to help the scientists to select the areas they would like to capture. It

projects a circle onto the first incident surface around the target point, at a radius proportional to the field of view as shown in Figure 4.6.

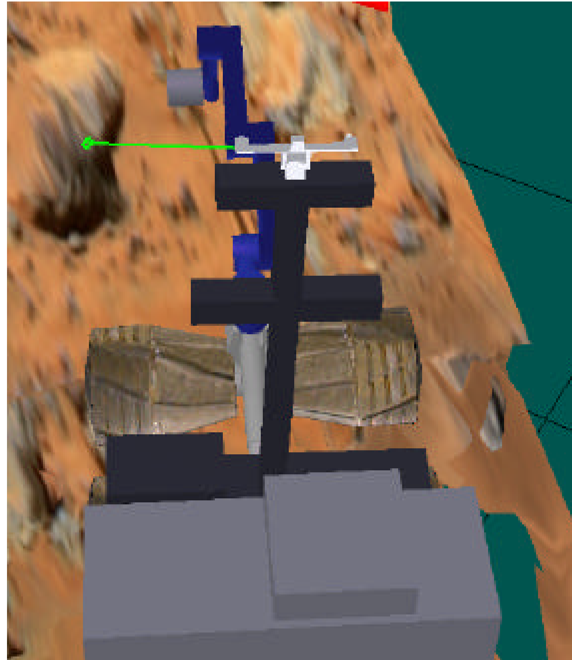

Figure 4.6 *Spectrometer Visualization Tool shown capturing a rock*

Since the circle is projected onto the first incident surface, it seems to adhere to the imaging surface. This graphically depicts the entire area being imaged. While the type of surface shown in Figure 4.7 is not uncommon, there is one additional condition under which this tool helps enhance visualization. It is possible that a distant object is being captured and there is some obstruction in the field of view of the spectrometer. Figure 4.8 depicts how the tool behaves when a local obstruction, i.e., the robot itself, interferes with the imaging field. If this occurs, the scientist can adjust the rover (or spectrometer) so the data gathered will be from the proper surface.

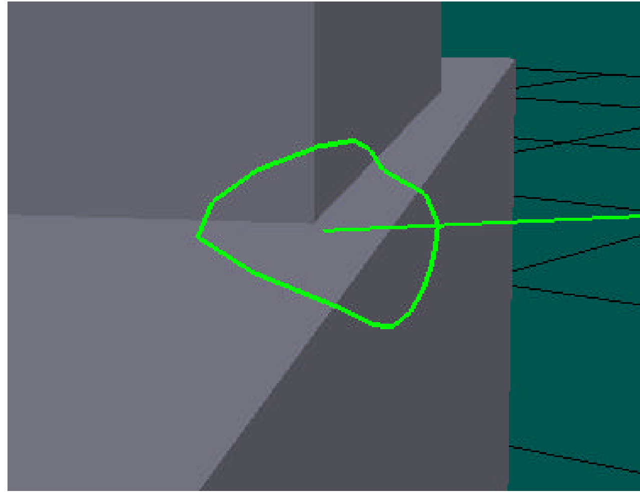

Figure 4.7 *Spectrometer Visualization Tool shown bending around the corner of a box*

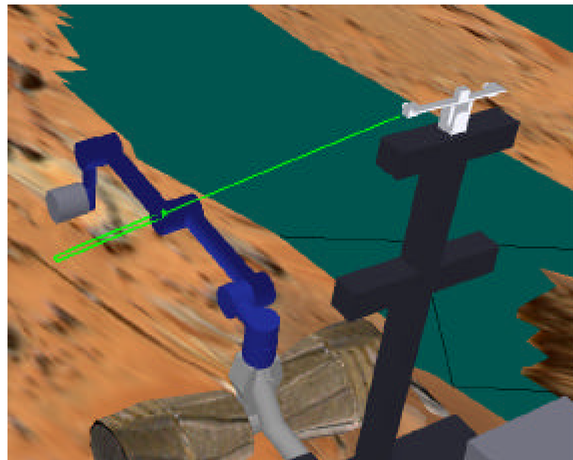

Figure 4.8 *Spectrometer Visualization Tool showing capture area with an obstruction*

The mathematics for the implementation of the spectrometer visualizer are difficult, but the concept is quite easy to understand. First, a line is drawn from the origin to the center of the target. This denotes the center of the acquisition region. Then, following a unit circle around the point of origin, the distance to the land is calculated. Once each point is found around the circle, the points are connected to draw the incident region.

4.2.5 Arm Task Planner

Planning an arm experiment is much like planning a rover task. Since the arm can do many experiments from one location, it is necessary to instance the arm for each experiment. Figure 4.9 shows the interface developed for the arm planner.

Figure 4.9 *Arm Task Planning Panel*

The graphical window acts as a navigational window for placing the arm in the proper position. The "Add Move" button instances the robot arm thus creating a new experiment.

Chapter 5

Results & Conclusions

5.1 Accomplishment of Project Goals

The goals set forth in this thesis were achieved, though not exactly as had been initially planned. The articulation of the Marsokhod rover was accomplished in three phases. The first phase served to integrate the rover into the MarsMap environment so it may be driven around the interface. Mathematical and physical assumptions were made to accelerate this integration process. The result of this phase was a fully articulated rover which emulated the physical nature of the Marsokhod rover. At this point, the accuracy of the virtual reality model was inadequate for the rigorous planning that was envisioned.

In the next phase, some of the mathematical assumptions made in the first phase were eliminated. The first refinement was to make the rover's mast properly articulate since it was previously assumed that the mast was rigid and always perpendicular to the ground. Another problem which was eliminated at this point was the fact that the complete visualization system would crash whenever the rover would drive over a "hole" in the terrain. This problem was solved by having the rover ignore the holes and proceed following the terrain as if it was on even land.

From the first phase of rover integration, there was a significant problem which plagued the simulation. When the rover would reach terrain with large height differences, it would enter an unstable oscillatory condition. Upon reaching this state, three things could be expected. One, the

rover could simply oscillate at some initial amplitude (frequency was based on the speed of the simulation loop and, therefore, remains constant for all three cases). In another case, considered the most desirable case, the rover would oscillate at a decreasing amplitude, ultimately settling into one resting position. The last case, considered to be the worst, resulted in the rover oscillating at an increasing amplitude. The typical result of this last case was a terminal loading of the visualization system to a point where it had to be restarted.

The problem was solved by imposing some real-world limitations on the previously unconstrained rover. The problem of the oscillation (as previously described) resulted from the robot trying to change heights too quickly. Many times this was a very transient problem, lasting while the rover drove over a rock or other sharp feature. The problem was solved by prohibiting the rover from changing its height instantaneously. An intermediate variable was added so the robot could only change its height incrementally until it reached the desired height.

5.2 Validation at the Field Test

The field test presented an excellent opportunity to test the planning tools implemented in this system. Due to problems incurred by the rover support team, there was less operational rover time than had been previously expected. Since time became a premium, there was less time to utilize the MarsokhodVR system. At one point in the mission, however, the scientists needed to take a panorama imaging sequence. It was at this point they first used the panorama camera planning tool. This interface brought a great deal of excitement to the scientists using it. Unlike before, they were able to preview, firsthand, the images the rover would be capturing and sending back. This provided them with the ability to capture the data they needed and eliminate ancillary imaging. The value of this tool was immediately recognized by the mission geologists using the tool.

5.3 Modularity

The most ambitious of all the goals set out to be accomplished was modularity of the system. While the system is not locked into simulating one robot, the interface is not totally modular. The system is modular to the extent that a complete software overhaul does not have to be done, and that other rovers can be integrated with minor code changes.

As mentioned above, the system is not fully modular. All the planning tools, while not exclusive to the Marsokhod rover, have parameters set to those on the Marsokhod rover. To add or remove a tool, a significant amount of reprogramming would be needed. Adding additional rovers would also require more work. First, the kinematics, and subsequently, the terrain following algorithms, for the new rover would have to be incorporated. The results would then have to be implemented into the visualization system.

5.4 Collision Detection

While several strategies for collision detection have been considered and discussed, collision detection was not implemented in this system. Suggested methods to implement collision detection are discussed in Chapter 6, under Future Work.

5.5 Discussions

As virtual reality becomes a more feasible technology, individuals and companies will undoubtedly wish to exploit it for high-level visualization systems. The work encompassed by this thesis has unearthed several issues which must be considered for these systems.

As with any large software design project, a proper method of software engineering should be selected. While it seems faster and easier to grow software, or keep building upon what exists until the final product is reached, it can lead to complications and frustrations toward the middle and end of the development process. Significant frustrations are often reached when there is no defined goal, or end, of the design.

If an official software engineering process is not adopted, it would be of significant benefit to create a requirements specification. The IEEE Standard 830, Software Requirements Specifications [8], provides a breakdown of the contents of the document. When designing a user interface, it is particularly important to expand on section 3.2.1 (External Interface Requirements, User Interfaces). The standard suggests to provide simulated screen layouts of the appearance of the user interface. This “pre-design” that goes into the project allows more programmers to work at once toward a common goal.

Projects like this also emphasize the need for physical modeling in virtual reality systems. Translating the physical world into a computer simulation is an ambitious task, and without a sound background in mathematics, it is difficult to take this step. This rigorous mathematical requirement demands highly trained programmers. For virtual reality simulation systems to be more accessible to the public, a higher level of kinematic specification needs to be implemented. If a programmer could simply specify the degree of freedom constraints, mass, and solidity for a collection of geometric objects which could be treated properly in a simulation, virtual reality will be a much easier tool to use.

During the field test, scientists wanted to test the robotic arm that is scheduled to go on the 2001 rover. With a good physics engine and modular rover interface, this could have been tested in virtual reality, as well. Not only that, but once added to the rover, the user interface could work with that particular rover and that particular arm. If the combination is found not to work, another combination could be tried in virtual reality before costly refitting is done to mount the part on another robot. Battelle Electronics and Avionics Systems has developed a bomb robot

simulation which allows budget constrained agencies to test a robot with various attachments before purchase [9].

The cost of virtual reality systems overshadows its value. If scientists take the time to survey ways they can extend their capabilities for visualization, the importance of virtual reality will outweight the cost. Before Pathfinder, the geologists used two dimensional images to measure the size of various surface features. They were unable to change the viewpoint of the image for a better view or perspective of the feature. The three dimensionality of the system eliminated this problem. Thus, MarsMap revolutionized the way planetary science is conducted [1].

There are many other disciplines which could benefit significantly from the use of virtual reality visualization systems. These applications range from design and layout of mass transportation systems to construction site management to assembly line training and monitoring. The possibilities for using virtual reality are endless.

Chapter 6

Future Work & Suggestions

This chapter provides suggestions and ideas for future work in this area based on the experience of working with this and other virtual reality systems.

6.1 Collision Detection

Collision detection is an important requirement for realistic interaction with virtual reality environments. It dictates the rules for how objects react to and interact with one another when they intersect. In the physical world, it is not possible for two solid objects to be in the same place at the same time. In virtual reality, however, objects do not have the inherent characteristics of mass, density, and all of the other values which comprise a real object. These computer generated models are represented by a collection of three dimensional points, and those points being filled in by a color or bitmap. With no physical value, these surfaces can easily pass through each other, leaving no damage or sign of being there. This is useful for some type of simulations, but it makes the physical reality, collisions, difficult to implement.

There are basically three kinds of collisions that need to be addressed in rover simulation: collisions with the terrain and associated objects, collisions with other independent objects, and self-collisions.

6.1.1 Handling Collisions

The scene graph, described in Appendix A, allows for a hierarchical relationship of objects in the virtual reality simulation. It also allows the users to isolate specific subtrees for data processing, or operations on the subtree. When the terrain is loaded into the scene graph, it is placed in its own subtree. It is possible to load additional objects in this same subtree (i.e. the lander model has the same parent as the terrain models, primarily because it is used the same way as the terrain models in terrain following). The terrain and any object in the same subtree will be considered as a terrain object. Of course, subtrees can be confined to be just one node.

The ideal situation would be if the computer is able to instantly indicate when two (or more) polygons make contact and then indicate this to the program. Unfortunately, the current system does not have this capability. While it is not out of the question for a system to have this ability, it would create a significant amount of overhead for the processor since every object or polygon would have to be checked against each other object or polygon in each simulation loop. The number of tests that would have to be done for all objects can be calculated as follows:

$$intersection\ \ tests = \sum_{i=0}^{n-1} i$$

where n is the number of objects. Generally, a small subset (typically one, or any, moving set) of objects is being tested against all the others.

To perform collision detection, there are a couple of methods that can be used. The one which offers the most fidelity is polygon level collision detection. This should be reserved for places where accuracy is in high demand since this level is extremely computationally intensive. The next level is bounding box collision detection. If the objects in the simulation are predominantly cubic, this is an excellent solution. An additional means for detection, almost a happy medium between polygon and bounding box detection, is ray detection. In this form, a ray is extended from a single point in a particular direction and detects if the ray passes through a polygon in the

search tree. In each of these cases, nearly every single polygon (or other bounding boxes) must be considered. Since computers do not have an inherent understanding of space, collision detections are not localized to the area of the object under consideration — unless this feature is programmed into the system.

As mentioned above, the ray detection method is currently used for terrain following for the rover. Terrain following, discussed in chapter 4, is a very controlled version of collision detection. It is a very specific problem which was addressed separately for clarity. It only checks to see where the rover can be placed. It makes no provision for collision detection for the rover with other objects.

Imagine a large rock is sitting on the surface of the terrain. The most desirable result for the simulation would be for the rover to stop moving once it comes into contact with that object. For this to work, the software must determine what collisions are occurring. The ray detection method is not suitable for this. The ray method is for detecting very localized or directional collisions and cannot reasonably detect all possible collisions. The next best option is the bounding box. The bounding boxes of each object in the simulation can be defined by any subtree as described above — it can be as simple as one node, or encompass several nodes. Using the bounding box, a quick and coarse detection can be done to determine whether or not there is proximal contact. If so, then a more exhaustive polygon level detection can be done. Once colliding polygons are identified, a response can occur. This response may simply be that the rover stops moving in that direction, or a part of the rover could be damaged (a complex behavior). Therefore, polygon level detection can reasonably be used after a rough estimate has been developed from bounding box detection.

Chapter 3 briefly mentioned a benefit of using the “move and settle” method of terrain following. If implemented, it could provide a reasonable solution to the problems encountered in terrain following mentioned above. One of the downfalls to using the collision detection in this method, or any iterative settling technique, is that once the collision is detected, that means that the model

has been transformed into a state where it is in collision. Therefore, the model must be transformed back to its previous position. This can be handled in two different ways. The first follows the method described above — move; test; if collide, backup; if not, repeat. The other process requires changing the event order. In the virtual reality simulation software, the event order is very specific. By default, the simulation loop reads sensor input, calls a specific action function, updates the objects using the sensor data, performs associated tasks, and renders the universe [12]. This order can be edited so that the model is first updated and then the models are updated with the sensor data. This is particularly useful because the last position of the object can be stored. Then, if the object is found to be colliding with another after sensor updates, the last position can be restored. Both methods achieve effectively the same objective.

Colliding with non-terrain objects provides a challenge of its own. As mentioned before, collisions are typically tested against anything that is moving, the underlying assumption being that non-moving objects are in an equilibrium state with the environment unaffected by collisions. One problem that needs to be considered is if both non-terrain objects are moveable, are both detecting collisions against the other? If this is the case, a form of deadlocking or even mutual exclusion may occur.

When dealing with terrain objects, it is assumed that the terrain is rigid and any articulation over it does not affect it. (Clearly this would be an invalid assumption for a bulldozer simulation but is suitable for the rover simulation.) It is not logical to apply this assumption to non-terrain objects. One rover driving over another could cause damage or even get stuck. The software must then be programmed to handle contingencies. Consider the case where one rover drives onto the top of another one. When the rover in contact with the ground moves, the rover on top should move with the driving rover. The result that would be expected from a simple collision detection engine would be that the bottom rover would drive out from under the other rover. The rover which was previously atop the driving rover would then settle to the ground as the drive rover slides out from underneath. These type of physical interactions relate back to the discussion in chapter 4 about the extent to which the environment needs to be modeled. If two rovers are never going to

be in close proximity to each other, there is no real need to simulate their interactions.

If there is good reason to model multiple rover interactions, it would be difficult to model exactly how they interact together in a physical environment. To make the simulation reasonable, assumptions should be made to simplify the problem. One assumption which would allow multiple rovers to exist in one virtual environment with limited interaction problems would be to prohibit the rovers from crossing another rover's bounding box, or any arbitrary radius from each rover. This would eliminate modeling the interactions because there would be none. Another possibility is to use the relative sizes of the rovers. If one rover is significantly larger than the other, it will, in all likelihood, always drive over the other rover. If the smaller rover starts driving while the large one is atop it, it might be reasonable for the smaller rover just to pull out because the larger rover may be still in contact with the ground and be held back by its own weight. Another simplification would be to paralyze movement of the smaller rover by the apparent weight of the imposing rover.

Another issue to consider is when a rover is intersecting with another part of itself. The Marsokhod rover has several possibilities for self intersection. The first condition which should be addressed goes back to the installation of certain hard stops that the rover has. With a physically accurate model of the rover, there should be no self intersecting of major rover parts (i.e. the axles and electronics boxes), unless the rover was poorly designed. If hard stops are not used, it may be wise to check to see if the new position is satisfactory, or else, to leave it in its previous position. On Marsokhod, the arm could very possibly intersect with the rover. A wise approach would be to check the arm against the rover. If the arm is hitting the rover, do not let it transform.

6.1.2 Improvements in Collision Detection

A better collision detection scheme, and hence, better terrain following will ultimately stem from

using a significantly more sophisticated physics engine. Many video games do a very good job of faking these physics but only on a very specialized scale. The physics programmed into video games are, for the most part, unique to that game. A generic physics engine could help solve many of these problems. Again, the current limits seem bound by processor speed. Until the time when such things are possible, there are several things which can be done.

Since one of the problems is that there is too much data to compare against for collisions, the number can be intelligently reduced by proximity. For a small amount of processor overhead, level-of detail nodes can be added. These nodes allow for various resolutions of models to be loaded into memory and rendered at the resolution which corresponds to the distance from the model to the viewpoint. One of the most effective uses of this node, yet not the most obvious, is to have no object loaded for distances greater than a specified amount. Therefore, if the user is not really close, there is no model to be tested. This could significantly reduce the number of collisions that need to be tested.

Another solution goes back to using bounding boxes. Instead of a simple bounding box collision detection on each object, a hierarchical bounding box collision detection scheme could be set up. For example, a rover's bounding box, if the rover is placed on the terrain, would be intersecting with the bounding box of the terrain. Then, the rover's robot arm could be tested against a rock. If the arm is colliding, the rover's position may need to be adjusted. On the other hand, if the bounding box of the rover is not intersecting with the terrain's bounding box, there is no reason to perform any further testing. This hierarchical approach directs a localized collision detection search. For some specific cases, it may take longer compared to a straight search, but on average, this technique should yield a reasonable performance increase.

In addition to bounding boxes, there is some work currently in progress that involves approximating objects with fundamental shapes, such as spheres [10]. This would even increase the fidelity of a system as simple as bounding box testing by letting the system test where the contact is most localized.

6.2 Improvements in Terrain Following

There are some areas in terrain following that require some attention. Of primary importance is developing a general method for terrain following. It may be possible to accomplish this by using a successive settling algorithm which would iteratively adjust the rover until it was settled on the terrain. This approach is very processor intensive and thus costly.

It is also useful to start considering what complexities can be added with respect to simulating the entire rover. Hard stops and other movement limitations should be able to be programmed. It would even be feasible to consider power monitoring in a very limited manner. Considering all of the additional subsystems would become very computationally intensive, but considering power consumption due to the motors and on-board instruments (i.e. the pan-tilt head, which is just a pair motors), would be a reasonable approximation.

The most significant problem is that the system developed is not general enough, and each time a new robot is to be simulated, an entirely new set of kinematics needs to be worked out. One solution currently under consideration is a file format which will contain all of the hierarchical and articulation data. This file would have to be loaded by a special program and would be associated with a set of functions which would be able to interpret and manage the articulation. With this, a library of parts could be included which could have power consumption models built-in and any additional features which would be useful to simulate. This would allow for quick prototyping and design of virtual, as well as, real rover systems.

Bibliography

[1] Ted Blackmon, Private Communication

[2] G. Burdea and P. Coiffet, *Virtual Reality Technology*, pp 5-7, John Wiley & Sons, Inc., 1994

[3] M. Mecham, "'Supervised Autonomy' Pays Dividends From Mars", *Aviation Week & Space Technology*, McGraw Hill, August 11, 1997

[4] C. Stoker, E. Zbinden, T. Blackmon, B. Kanefsky, J. Hagen, P. Henning, C. Neveu, D. Rasmussen, K. Schwehr, and M. Sims, "Analyzing Pathfinder Data using Virtual Reality and Super-resolved Imaging", *Journal of Geophysical Research*, Special Mars Pathfinder Issue

[5] C. Stoker, T. Blackmon, J. Hagen, B. Kanefsky, D. Rasmussen, K. Schwehr, M. Sims, and E. Zbinden, "MarsMap: An Interactive Virtual Reality Model of the Pathfinder Landing Site"

[6] C. Allport, B. Schreiner, P. Sines, and B. Das, Virtual Reality Semiconductor Laboratory: An Advanced Training Tool for Teaching Complex Ideas, 1999 International Conference on Visual Computing, (Goa, India March 1999)

[7] H. Rheingold, *Virtual Reality*, pg. 16, Simon & Schuster, 1991

[8] H. van Vliet, *Software Engineering*, pp. 515-518, John Wiley & Sons, Inc., 1993

[9] J. Lindwall, "Sense8 News & Product Update", March 1999

[10] P. Hubbard, "Time Critical Collision Detection", Cornell University, http://www.cs.brown.edu/stc/education/course95-96/TC-Collision-Detection/

[11] EAI Corporation, Mill Valley, California, *http://www.sense8.com*

[12] Sense8 Corporation, *WorldToolKit Reference Manual*, pp 3.15-18, April 1997

[13] C. S. Allport, B. D. Schreiner, and P. B. Sines, Virtual Reality Semiconductor Laboratory, IEEE 1997 Frontiers in Education Conference Program, p. 58, (Pittsburgh, PA November 1997)

This research received media coverage as listed below:

Media Service, Location	**Date**
West Virginia University Media Services, Morgantown, WV	February 1999
The Herald, Hagerstown, MD	February 18, 1999
The Martinsburg Journal, Martinsburg, WV	February 1999
The Charleston Gazette, Charleston, WV	February 19, 1999
Times West Virginian, Fairmont, WV	February 19, 1999
The Dominion Post, Morgantown, WV	February 20, 1999
The Daily Athenaeum, West Virginia University, Morgantown, WV	March 2, 1999
The Intermountain, Elkins, WV	March 1999
The Moundsville Echo, Moundsville, WV	March 1999
The Parkersburg Sentinel, Parkersburg, WV	March 1999
KDKA, Pittsburgh, PA	March 12, 1999

Appendix A

Explanation of Virtual Reality Concepts

This purpose of this appendix is to provide some background information regarding virtual reality concepts. Intermingled with the general virtual reality information will be some discussion about how various utilities are implemented by EAI's (formerly Sense8 Corporation) WorldToolKit [11]. This review is not meant to be exhaustive in any way. It is simply meant to provide enough understanding of the challenges faced by this project and the concepts applied.

A.1 The Scene Graph

A scene graph is a data structure which provides a hierarchical organization for objects in a virtual reality simulation. It inherently provides an organized explanation for how each object in the scene is to be drawn, illuminated, and otherwise transformed. This data structure is typically implemented as a tree and traversal is usually done in a depth-first manner for reasons which will soon be apparent.

In a scene graph, there are many types of nodes which can be added, but there are only five nodes necessary for understanding the basic operation of a scene graph. The first type of node is a root node. All tree implementations have a root node, the topmost node in the tree, to which all other nodes attach. There is only one root node per scene graph.

Another type of node is a light node. This node has different properties which specify how the objects it acts upon are to be illuminated. (Each object, as well, has information about how it

should respond to the light.) Construction of a scene graph cannot be done haphazardly. If not done correctly, the resulting scene will not be representative of what the user hoped to see. One of the most important pitfalls to be wary of in constructing a scene graph is that in order for any type of transform to be applied to an object (or the rest of the scene), the nodes to be affected must appear after the node which holds the transform information [12]. Figure A-1 shows some different node relationships to illustrate this concept.

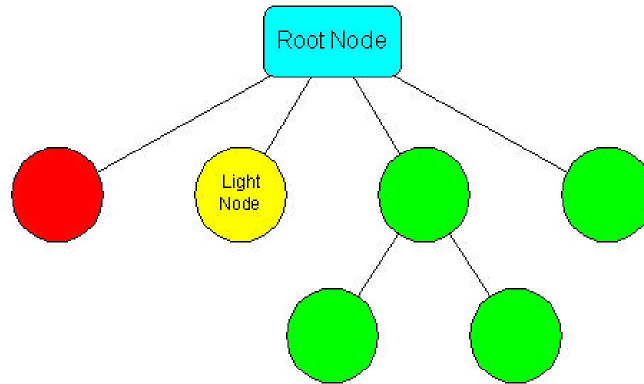

Figure A-1 *Sample Scene Graph - all of the nodes in green are affected by the light node (they come after the light node) whereas the red node is not (it comes before the light node)*

A virtual environment would not be very interesting without objects in it, so another type of node, a geometry node, is provided for the objects. These nodes simply hold the geometry information of the objects. Incorporated with this geometry information is the material information for the object. The material information is all of the information about how the surface is to be illuminated by applied light.

In order to actually move and rotate an object, another type of node is needed. This type of node is a transform node. It contains all of the position and orientation information for the nodes it acts upon. Transform nodes follow the same transform effect rules as the light node does (refer to Figure A-1). It is possible that the user will want to limit the transform applied by the transform and light nodes. This ability is provided for in the last type of node to be discussed here.

Separator nodes prevent transforms, whether light or rotation-translation transforms, from propagating up and across the tree. Figure A-2 shows a sample scene graph with separator nodes added.

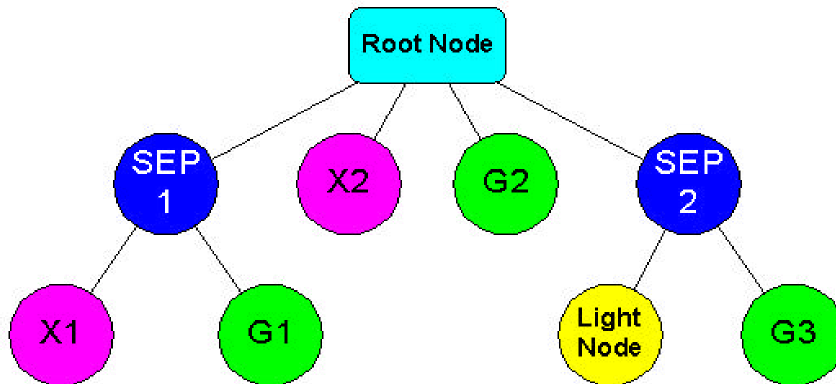

Figure A-2 *Separator Nodes isolate transforms and keep them from affecting the rest of the scene. The blue nodes are separator nodes, the violet are transform nodes, the green are geometry nodes, and the yellow is a light node.*

In this figure, separator node SEP-1 keeps transform X1 from acting on anything except for geometry G1. Transform X2 affects geometry G2 and everything else after it in the scene since it is not isolated in any way. Separator SEP-2 does not isolate transform X2 because it is not hierarchically above X2. It does, however, isolate the light node to only illuminate geometry G3.

Scene graphs are not static constructs. During the simulation, many nodes are added to and removed from it. The ability of the scene graph to change like this empowers the programmer to make a more interactive virtual reality environment.

A.2 Reference Frames

In any interactive virtual reality simulation, something moves. Whether it be an object or a viewpoint, something in the simulation is moving. Given the fact that virtual reality is three-

dimensional, the direction in which the thing is moving has components in each of the three directions. Tracking these movements as vectors and calculating new positions accordingly is extremely tedious work. It is difficult in keeping track of the information and in the mathematics — especially when it comes to rotations. It is for these reasons that various frames of reference are used in VR. WorldToolKit provides four different frames, three of which will be described below.

The coordinate system used by WorldToolKit follows the right-hand rule. The positive Y-axis points straight down (as opposed to up which may be expected). Imagine standing at the origin of the coordinate frame, the positive Z-axis would be extending straight out in front and the positive X-axis would extend to the right. The coordinate system used is Cartesian.

The first reference frame is the world frame. These are the basic, untransformed, axes in a simulation. The +X- and +Z-axes can be thought of as the cardinal directions (east and north, respectively) for everything in the simulation.

The viewpoint frame is the reference frame which corresponds to the position of the viewpoint. No matter what direction the viewpoint is facing (with respect to the world frame), the +Z-axis is always forward. This allows the programmer to move the viewpoint forward by simply translating along the +Z-axis. In contrast, if the only frame available was the world frame, the programmer would have to get the orientation of the viewpoint, calculate the corresponding direction, add the starting location, and translate along that direction vector just to move the viewpoint. A comparable frame is provided for objects.

Each object that is loaded has a local frame. This is a frame of reference in which the object can move “forward” by just translating along its +Z-axis. Forward for the object is determined by the orientation it had when it was created in an external program. For example, a car would be drawn such that the headlights pointed down the +Z-axis. To really emphasis the use of these frames of reference, consult Figure A-3.

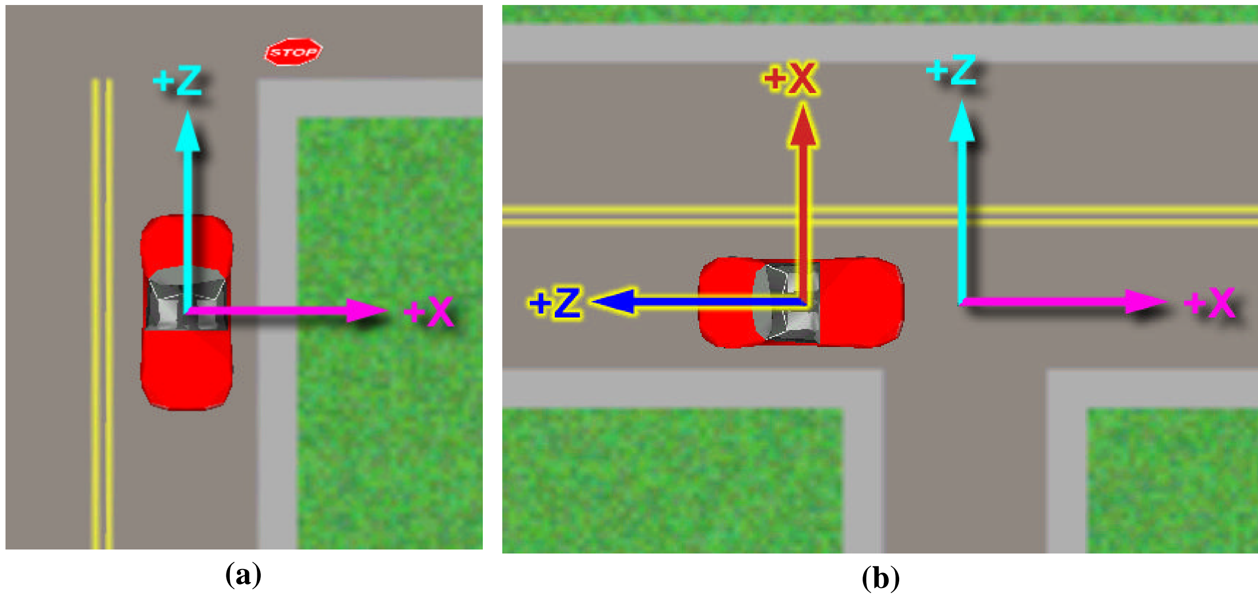

Figure A-3 *Illustration of object moving along local and world axes. The cyan-violet axes represent the world frame and the blue-red axes represent the local axes. In (a), the axes are aligned, and only the world axes are shown.*

In the first image, the car's direction is aligned with the world axis, or world frame, and moving the car in the +Z direction would result in the car going forward (in both frames). In the second image, the car is now pointing down the world's negative X-axis. Moving the car down the +Z-axis in the world frame would result in the car sliding to its right. Using its local frame, the car would move forward as it is translated down its local +Z-axis.

A.3 Progressive Transforms

One critical aspect of transforms was omitted in the discussion about scene graphs. The omission was intentional because this topic requires a discussion of its own. In the discussion above, it was mentioned that any transforms would be applied to any node after the transform. It is also true that other transform nodes at a later point in the scene graph are affected. Figure A-4 shows a typical example of this occurrence. In this image, geometry G1 is rotated 120° around the X-axis

as specified by transform X1. Geometry G2 is rotated by -30° around the X-axis as specified by transform X2 *and* by 120° as specified by transform X1. The ultimate transform for G2 is -30° + 120° = 90° around the X-axis.

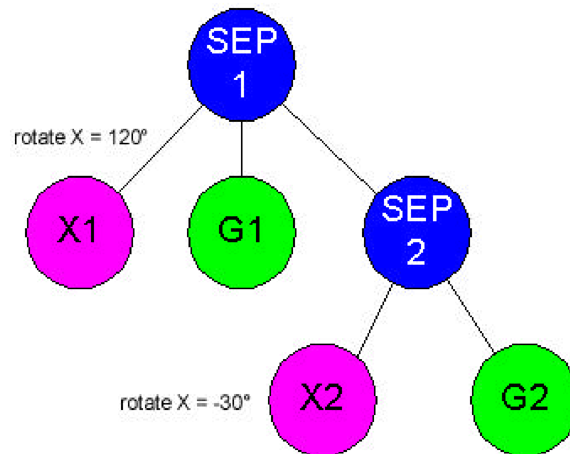

Figure A-4 *Progressive Transform — The blue nodes are separator nodes, the violet are transform nodes, and the green are geometry nodes. The violet nodes could just as well be light nodes providing a progressive lighting effect.*

Appendix B

Pan-Tilt Sequence Planner Source Code

This is a complete listing of the source code (in C) for the pan-tilt sequence planner. After the primary implementation by Allport, additions were made by Ted Blackmon, head of rover visualization, Intelligent Mechanisms Group, NASA-Ames, Mountain View, California. Blackmon's changes are included.

B.1 Listing of panCam.h

```
/****************************************************************************
** panCam.h - planning module for panorama Image camera for Marsokhod rover
****************************************************************************/
void panCam_init();
void panCam_loop();
void panCam_exit();
WTnode *panCam_newTask(WTnode *);
void panCam_editTask(WTnode *);
void panCam_seq(WTnode *node, FILE *file);
void panCam_displayMosaic(WTnode *, char *filepath);
```

B.2 Listing of panCam.c

```
/****************************************************************************
** panCam.c - planning & display module for panorama Image camera
**        on the Marsokhod rover
**
** Written by Theodore T. Blackmon & Chris Allport
** Copyright 1999.
****************************************************************************/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <assert.h>

#include "wt.h"
#include "ims.h"
#include "image.h"
#include "overlay.h"
#include "text3d.h"

#include "rover_plan.h"
// #include "panCam_sequence.h"
#include "marsokhod.h"

/*** These need to be variables, not hardcoded ***/
char imageFileBasename[2048];
char crlFilename[2048];
/*********************************************************/

/*** PanCam planning functions ***/
void panCam_init();
void panCam_loop();
void panCam_exit();
WTnode *panCam_newTask(WTnode *marso);
void panCam_editTask(WTnode *node);
void panCam_seq(WTnode *node, FILE *crlFile);

/*** Map data functions ***/
//void panCam_displayImages(WTnode *, char *filepath);
void panCam_displayMoasicDome(WTnode *, char *filepath);
void panCam_displayMosaicPlanimetric(WTnode *node, char *filepath);

/*********************************************************/

typedef struct {
  float pMin;
  float pMax;
  float pRes;
  float tMin;
  float tMax;
  float tRes;
  int camID;
```

```
  int resolution;
  int compression;
  WTpq camera_pose;

} PanCamInfo;

/*** Camera Definitions ***/
#define SCI_CAMERA_HRC_LEFT     0              /* high res color */
#define SCI_CAMERA_HRC_RIGHT    1
#define SCI_CAMERA_HRC_STEREO   3
#define SCI_CAMERA_WM_LEFT      4              /* wide angle monochrome */
#define SCI_CAMERA_WM_RIGHT     5
#define SCI_CAMERA_WM_STEREO    7
#define SCI_CAMERA_HRR_LEFT     8              /* high res red */
#define SCI_CAMERA_HRR_RIGHT    9
#define SCI_CAMERA_HRR_STEREO   11
#define SCI_CAMERA_HRG_LEFT     12             /* high res green */
#define SCI_CAMERA_HRG_RIGHT    13
#define SCI_CAMERA_HRG_STEREO   15
#define SCI_CAMERA_HRB_LEFT           16             /* high res blue */
#define SCI_CAMERA_HRB_RIGHT    17
#define SCI_CAMERA_HRB_STEREO   19

#define HIRES_CAM           0
#define LORES_CAM           1
static int panCam_type=HIRES_CAM;
static char panCamId_text[256];

/*** Default Definitions ***/
#define PAN_MIN                     -89
#define PAN_MAX                     258
#define TILT_MIN            -88
#define TILT_MAX            76
#define HIRES_FOVX                   180.0f*(0.191136)/3.14159f
#define HIRES_FOVY                   180.0f*(0.143352)/3.14159f
#define LORES_FOVX                   180.0f*(0.608640)/3.14159f
#define LORES_FOVY                   180.0f*(0.456480)/3.14159f

#define PAN_MIN_DEFAULT                    -30
#define PAN_MAX_DEFAULT                     30
#define PAN_RES_DEFAULT                     10
#define TILT_MIN_DEFAULT       -25
#define TILT_MAX_DEFAULT        25
#define TILT_RES_DEFAULT        8
```

```
#define CAMERA_DEFAULT                    3
#define RESOLUTION_DEFAULT       1
#define COMPRESSION_DEFAULT      8

/*** pan-cam nodes ***/
static WTnode *panCam_taskNode;
static WTnode *panCam_node;

/*** Pan Cam UI ***/
static WTui *pwtuiPanCam;
static WTui *pwtuiPanResSlider, *pwtuiTiltResSlider;
static void BuildPanCamUI(WTui *pwtui);
static void PanCamSelectCamera(WTui *pStruct, void *pData);
static void PanCamSelectRes(WTui *pStruct, void *pData);
static void PanCamSelectComp(WTui *pStruct, void *pData);
static void PanCamPanResSlider(WTui *pStruct, void *pData);
static void PanCamTiltResSlider(WTui *pStruct, void *pData);
static void PanCamPanMinSlider(WTui *pStruct, void *pData);
static void PanCamPanMaxSlider(WTui *pStruct, void *pData);
static void PanCamTiltMinSlider(WTui *pStruct, void *pData);
static void PanCamTiltMaxSlider(WTui *pStruct, void *pData);
static void PanCamResetSliders(int panCamType);
static WTui *pwtuiPanMinSlider, *pwtuiTiltMinSlider, *pwtuiPanMaxSlider,
*pwtuiTiltMaxSlider;

/*** 2D & 3D overaly functions for panCam planning ***/
static void panCam_image_projections(WTwindow *w, FLAG eye);
static void panCam_plan_readout(WTwindow *w, FLAG eye);
static void calculate_3d_point(float pan, float tilt, WTp3 p);
static float fPanCamProjectionDistance = 2.0f;
static float fPanCamRadius = 5.0f;

/*** Variables for time and data volume stats ***/
static int pan_count, tilt_count;                /*** number of pan/tilt increments in a panorama
***/
static float panCam_estTime;                     /*** in minutes ***/
static float panCam_estDataVol;                         /*** in mBits ***/
#define PANCAM_TIME_PER_PT               0.5f
#define PANCAM_MBITS_PER_IMAGE                  2.4

/*****************************************************************
**  Initialization function for pan cam planning
*****************************************************************/
```

```
void panCam_init()
{
	WTui *ui;


	sprintf(imageFileBasename,"21Jan99/downlink1/images/ip3_s001/ip3_s001");
	sprintf(crlFilename,"ip3_s001.crl");

	WTmessage("Pan Cam Init\n");
	/*** activate rover planning tool ***/
	tool_activate(roverPlan_control, TRUE);

	/*** Build planning UI for Pan Cam Imager ***/
	ui = BuildGeneralUI();
	BuildPanCamUI(ui);
	WTui_manage(ui);

	/*** add 3d overlay for panorama image projection window ***/
	overlay3d_add(panCam_image_projections);
	/*** add 2d overlay for panorama image information ***/
	overlay2d_add(panCam_plan_readout);
}

void panCam_loop()
{
}

void panCam_exit()
{
	/*** remove 3d overlay for panorama image projection window ***/
	overlay3d_delete(panCam_image_projections);
	/*** remove 2d overlay for panorama image information ***/
	overlay2d_delete(panCam_plan_readout);
	WTmessage("Leaving Pan Can\n");
}

WTnode *panCam_newTask(WTnode *marso)
{
	WTnode *xform;
	PanCamInfo *info;
	WTp3 pos;

	WTmessage("Create a new pan-cam node!\n");
	/*** make a sep, xform, and geom node ***/
```

```
	panCam_taskNode = WTsepnode_new(NULL);
	xform = WTxformnode_new(panCam_taskNode);
	pos[X] = 0.0f;
	pos[Y] = 0.0f;
	pos[Z] = -1.6f;
	WTnode_settranslation(xform, pos);
	panCam_node = WTgeometrynode_new(panCam_taskNode,
			WTgeometry_newsphere(0.5, 8, 8, FALSE, TRUE));
	WTnode_enable(panCam_node, FALSE);

	/*** allocate memory for pan-cam info struct ***/
	info = (PanCamInfo *)malloc(sizeof(PanCamInfo));
	if(info == NULL) printf("malloc failed for PanCamInfo struct.\n");
	info->pMin = PAN_MIN_DEFAULT;
	info->pMax = PAN_MAX_DEFAULT;
	info->pRes = PAN_RES_DEFAULT;
	info->tMin = TILT_MIN_DEFAULT;
	info->tMax = TILT_MAX_DEFAULT;
	info->tRes = TILT_RES_DEFAULT;
	info->resolution = RESOLUTION_DEFAULT;
	info->compression = COMPRESSION_DEFAULT;
	info->camID = SCI_CAMERA_HRC_STEREO;
	sprintf(panCamId_text, "Hi-Res, Stereo Color");

	/*** Set Rover Camera Position ***/
	marsokhod_panCam_xform(WTnode_getchild(marso,0), &info->camera_pose);

	/*** set default value of pan-cam info struct ***/
	WTnode_setdata(panCam_node, (void *)info);

	return(panCam_taskNode);
}

void panCam_editTask(WTnode *node)
{
	WTmessage("Edit pan-cam node!\n");
	panCam_node = WTnode_getchild(node, 1);
}

void panCam_seq(WTnode *node, FILE *crlFile)
{
	PanCamInfo *info;

	panCam_node = WTnode_getchild(node, 1);
```

```
	info = (PanCamInfo *) WTnode_getdata(panCam_node);

	fprintf(crlFile, "      :featureimage \"\"\n");
	fprintf(crlFile, "      :featurepx %f\n",-1.0);
	fprintf(crlFile, "      :featurepy %f\n",-1.0);
	fprintf(crlFile, "      ;for pancam\n");
	fprintf(crlFile, "      :camid %d \n",info->camID);
	fprintf(crlFile, "      :pmin %f\n",radians(info->pMin));
	fprintf(crlFile, "      :pmax %f\n",radians(info->pMax));
	fprintf(crlFile, "      :pres %f\n",radians(info->pRes));
	fprintf(crlFile, "      :tmin %f\n",radians(info->tMin));
	fprintf(crlFile, "      :tmax %f\n",radians(info->tMax));
	fprintf(crlFile, "      :tres %f\n",radians(info->tRes));
	fprintf(crlFile, "      :resolution %d \n",info->resolution);
	fprintf(crlFile, "      :compression %d \n",info->compression);
	fprintf(crlFile, "      :basefilename \"\"\n");


	printf("PanCam sequence ...\n");
	printf("      :featureImage \"\"\n");
	printf("      :featurePx %f\n",-1.0);
	printf("      :featurePy %f\n",-1.0);
	printf("      ;for panCam\n");
	printf("      :camId %d \n",info->camID);
	printf("      :pMin %f (%f deg)\n",radians(info->pMin), info->pMin);
	printf("      :pMax %f (%f deg)\n",radians(info->pMax), info->pMax);
	printf("      :pRes %f (%f deg)\n",radians(info->pRes), info->pRes);
	printf("      :tMin %f (%f deg)\n",radians(info->tMin), info->tMin);
	printf("      :tMax %f (%f deg)\n",radians(info->tMax), info->tMax);
	printf("      :tRes %f (%f deg)\n",radians(info->tRes), info->tRes);
	printf("      :resolution %d \n",info->resolution);
	printf("      :compression %d \n",info->compression);
	printf("      :baseFileName \"\"\n");
}

/********************************************************************
** Create an image dome model using a mercator projection panorama
********************************************************************/
void panCam_displayMosaicDome(WTnode *node, char *filepath)
{
	Image_Dome_Info *dome_info;
	PanCamInfo *info;
	WTnode *panCam_dome;
	WTnodepath *nodepath;
```

```
	WTpq pose1, pose2;
	char *filename;
	char panImage_filename[256];
	char panImage_dateTime[256];
	char panImage_basename[WTPATHLEN];
	WTdirectory *dir;
	FILE *file;
	char *fname;
	char *parse_fname;
	char path[WTPATHLEN];
	char *period;
	char *plusChar,*nameChar;
	char *offsetChar1,*offsetChar2,*offsetChar3,*offsetChar4;
	char *date;
	int px,py;
	int row_size, col_size;
	int subrow_size, subcol_size;
	int subrow_offset, subcol_offset;
	float pMin, pMax, tMin, tMax;
	float pan, tilt, fovx, fovy;
	char *lastbackslash;

	WTmessage("Display pan-cam node!\n");
#ifdef SGI
	filename = strrchr(filepath, '/');
#endif
#ifdef NT
	filename = strrchr(filepath, '\\');
#endif
	filename = filename++;
	WTmessage("filename : %s\n",filename);

	/*** allocate memory for pan-cam info struct ***/
	info = (PanCamInfo *)malloc(sizeof(PanCamInfo));
	if(info == NULL) printf("malloc failed for PanCamInfo struct.\n");

	/*** open sub panorama image header file ***/
	file = fopen(filepath, "r");
	fscanf(file,"#name=%s\n",panImage_filename);
	fscanf(file,"#date=%[^\n]\n",panImage_dateTime);
	fscanf(file,"#row_size=%d\n",&row_size);
	fscanf(file,"#col_size=%d\n",&col_size);
	fscanf(file,"#panMin=%f\n",&pMin);
	fscanf(file,"#tiltMin=%f\n",&tMin);
```

```
	fscanf(file,"#panMax=%f\n",&pMax);
	fscanf(file,"#tiltMax=%f\n",&tMax);

	printf("#name=%s\n",panImage_filename);
	printf("#date=%s\n",panImage_dateTime);
	printf("#row_size=%d\n",row_size);
	printf("#col_size=%d\n",col_size);
	printf("#panMin=%f\n",pMin);
	printf("#tiltMin=%f\n",tMin);
	printf("#panMax=%f\n",pMax);
	printf("#tiltMax=%f\n",tMax);

	/* create an image dome model */
	dome_info = (Image_Dome_Info *)malloc(sizeof(Image_Dome_Info));

	/*** Set Rover Camera Position ***/
	marsokhod_panCam_xform(node, &pose1);
	pose1.p[X] -= 0.125f;
	pose1.p[Y] += 0.01f;
	pose1.p[Z] += 0.05f;
	nodepath = WTnodepath_new(WTnode_getchild(node,0), ims_root, 0);
	WTnodepath_gettranslation(nodepath, pose2.p);
	WTnodepath_getorientation(nodepath, pose2.q);
	WTnodepath_delete(nodepath);
	WTpq_world2localframe(&pose1, &pose2, &dome_info->pose);

	/**** create new image dome node ***/
	panCam_dome = image_dome_new(dome_info);
	WTnode_addchild(node, panCam_dome);
	WTnode_enable(panCam_dome, TRUE);

	/*** set basename for image sub divisions ***/
	period = strrchr(panImage_filename, '.');
	*period = '\0';
	sprintf(panImage_basename,"%s",panImage_filename);

	/*** set path to image directory ***/
#ifdef SGI
	lastbackslash = strrchr(filepath, '/');
	WTinit_setimages(filepath);
#endif
#ifdef NT
	lastbackslash = strrchr(filepath, '\\');
#endif
```

```
/*** remove file selected to retain directory path ***/
*lastbackslash = '\0';
/*** open directory and read contents ***/
WTmessage("Open directory %s\n.",filepath);
dir = WTdirectory_open(filepath);
if (!dir) {
        WTui *noDirectoryMessageBox;
        char buf[256];
        sprintf(buf, "%s", filepath);
        noDirectoryMessageBox =
                WTui_newmessagebox(toplevel, buf, "Directory not found");
}
else   {
   WTmessage(">> Load images from directory %s.\n",filepath);
   while((fname=WTdirectory_getentry(dir)) != NULL)  {
        if(!strncmp(fname,panImage_basename,strlen(panImage_basename)))  {
           WTmessage("fname=%s\n",fname);
           period = strrchr(fname, '.');
           if( !strcmp(period, ".jpg"))  {
                /*** parse image subdivision pixel boundaries ***/
                WTmessage("file=%s\n",fname);
                strcpy(parse_fname, fname);
                nameChar = strstr(parse_fname, panImage_basename);
                nameChar = nameChar + strlen(panImage_basename);
                //WTmessage("\nbasename %s\n NAMECHAR:
                        %s\n\n",panImage_basename,nameChar);
                plusChar = strchr(nameChar, '_');
                *plusChar = '\0';
                offsetChar1 = plusChar+1;
                plusChar = strchr(offsetChar1, 'x');
                *plusChar = '\0';
                offsetChar2 = plusChar+1;
                plusChar = strchr(offsetChar2, '+');
                *plusChar = '\0';
                offsetChar3 = plusChar+1;
                plusChar = strchr(offsetChar3, '+');
                *plusChar = '\0';
                offsetChar4 = plusChar+1;
                plusChar = strchr(offsetChar4, '.');
                *plusChar = '\0';
                subrow_size = atoi(offsetChar1);
                subcol_size = atoi(offsetChar2);
                subrow_offset = atoi(offsetChar3);
                subcol_offset = atoi(offsetChar4);
```

```
				WTmessage("subrow_size=%d subcol_size=%d subrow_offset=%d
					subcol_offset=%d\n", subrow_size, subcol_size, subrow_offset,
					subcol_offset);

				/*** convert image boundaries to pan,tilt, fovx, fovy ***/
				if((subrow_offset+subrow_size)>row_size)
					subrow_size=row_size-subrow_offset;
				pan = pMin + ((subrow_offset+0.5f*subrow_size) /row_size) *
					(pMax-pMin);
				if((subcol_offset+subcol_size)>col_size)
					subcol_size=col_size-subcol_offset;
				tilt = tMin + ((col_size-subcol_offset-0.5f*subcol_size)/col_size)*
					(tMax-tMin);
				fovx = ((float)subrow_size/(float)row_size)*(pMax-pMin);
				fovy = ((float)subcol_size/(float)col_size)*(tMax-tMin);
				WTmessage("pan=%f tilt=%f fovx=%f fovy=%f\n",
					degrees(pan), degrees(tilt), degrees(fovx), degrees(fovy));

				/*** create a 3d image dome model wedge ***/
				strcpy(path, filepath);
				strcat(path, WTFILE_DELIM);
				strcat(path, fname);
				WTmessage("load image wedge fname: %s\n\n",path);
				image_dome_addM(panCam_dome, fPanCamRadius,
					degrees(pan), -degrees(tilt), degrees(fovx), degrees(fovy), path);
				WTuniverse_go1();
			}
		}
	}
	WTdirectory_close(dir);
	WTmessage("\n\n");
	}
}

/*******************************************************************
** Create an image dome model using a mercator projection panorama
*******************************************************************/
void panCam_displayMosaicPlanimetric(WTnode *node, char *filepath)
{
	Image_Planimetric_Info *p_info;
	PanCamInfo *info;
	WTnode *panCam_planimetric;
	WTnodepath *nodepath;
	WTpq pose1, pose2;
```

```
	char *filename;
	char panImage_filename[256];
	char panImage_dateTime[256];
	char panImage_basename[WTPATHLEN];
	WTdirectory *dir;
	FILE *file;
	char *fname;
	char *parse_fname;
	char path[WTPATHLEN];
	char *period;
	char *plusChar,*nameChar;
	char *offsetChar1,*offsetChar2,*offsetChar3,*offsetChar4;
	char *date;
	int px,py;
	int row_size, col_size;
	int subrow_size, subcol_size;
	int subrow_offset, subcol_offset;
	float pMin, pMax, tMin, tMax;
	float pan, tilt;
	float p1, p2, t1, t2;
	char *lastbackslash;

	WTmessage("Display pan-cam node!\n");
#ifdef SGI
	filename = strrchr(filepath, '/');
#endif
#ifdef NT
	filename = strrchr(filepath, '\\');
#endif
	filename = filename++;
	WTmessage("filename : %s\n",filename);

	/*** allocate memory for pan-cam info struct ***/
	info = (PanCamInfo *)malloc(sizeof(PanCamInfo));
	if(info == NULL) printf("malloc failed for PanCamInfo struct.\n");

	/*** open sub panorama image header file ***/
	file = fopen(filepath, "r");
	fscanf(file,"#name=%s\n",panImage_filename);
	fscanf(file,"#date=%[^\n]\n",panImage_dateTime);
	fscanf(file,"#row_size=%d\n",&row_size);
	fscanf(file,"#col_size=%d\n",&col_size);
	fscanf(file,"#panMin=%f\n",&pMin);
	fscanf(file,"#tiltMin=%f\n",&tMin);
```

```
	fscanf(file,"#panMax=%f\n",&pMax);
	fscanf(file,"#tiltMax=%f\n",&tMax);

	printf("#name=%s\n",panImage_filename);
	printf("#date=%s\n",panImage_dateTime);
	printf("#row_size=%d\n",row_size);
	printf("#col_size=%d\n",col_size);
	printf("#panMin=%f\n",pMin);
	printf("#tiltMin=%f\n",tMin);
	printf("#panMax=%f\n",pMax);
	printf("#tiltMax=%f\n",tMax);

	/* create an image dome model */
	p_info = (Image_Planimetric_Info *)malloc(sizeof(Image_Planimetric_Info));

	/*** Set Rover Camera Position ***/
	marsokhod_panCam_xform(node, &pose1);
	pose1.p[X] -= 0.125f;
	pose1.p[Y] += 0.01f;
	pose1.p[Z] += 0.05f;
	nodepath = WTnodepath_new(WTnode_getchild(node,0), ims_root, 0);
	WTnodepath_gettranslation(nodepath, pose2.p);
	WTnodepath_getorientation(nodepath, pose2.q);
	WTnodepath_delete(nodepath);
	WTpq_world2localframe(&pose1, &pose2, &p_info->pose);

	p_info->camHeight = 1.5;
	p_info->rMax = 25.0f;

	/**** create new image planimetric node ***/
	panCam_planimetric = image_planimetric_new(p_info);
	WTnode_addchild(node, panCam_planimetric);
	WTnode_enable(panCam_planimetric, TRUE);

	/*** set basename for image sub divisions ***/
	period = strrchr(panImage_filename, '.');
	*period = '\0';
	sprintf(panImage_basename,"%s",panImage_filename);

	/*** set path to image directory ***/
#ifdef SGI
	lastbackslash = strrchr(filepath, '/');
	WTinit_setimages(filepath);
#endif
```

```
#ifdef NT
        lastbackslash = strrchr(filepath, '\\');
#endif
        /*** remove file selected to retain directory path ***/
        *lastbackslash = '\0';
        /*** open directory and read contents ***/
        WTmessage("Open directory %s\n.",filepath);
        dir = WTdirectory_open(filepath);
        if (!dir) {
                WTui *noDirectoryMessageBox;
                char buf[256];
                sprintf(buf, "%s", filepath);
                noDirectoryMessageBox =
                        WTui_newmessagebox(toplevel, buf, "Directory not found");
        }
        else   {
           WTmessage(">> Load images from directory %s.\n",filepath);
           while((fname=WTdirectory_getentry(dir)) != NULL)  {
                if(!strncmp(fname,panImage_basename,strlen(panImage_basename)))  {
                   WTmessage("fname=%s\n",fname);
                   period = strrchr(fname, '.');
                   if( !strcmp(period, ".jpg"))  {
                        /*** parse image subdivision pixel boundaries ***/
                        WTmessage("file=%s\n",fname);
                        strcpy(parse_fname, fname);
                        nameChar = strstr(parse_fname, panImage_basename);
                        nameChar = nameChar + strlen(panImage_basename);
                        //WTmessage("\nbasename %s\n NAMECHAR:
                                %s\n\n",panImage_basename,nameChar);
                        plusChar = strchr(nameChar, '_');
                        *plusChar = '\0';
                        offsetChar1 = plusChar+1;
                        plusChar = strchr(offsetChar1, 'x');
                        *plusChar = '\0';
                        offsetChar2 = plusChar+1;
                        plusChar = strchr(offsetChar2, '+');
                        *plusChar = '\0';
                        offsetChar3 = plusChar+1;
                        plusChar = strchr(offsetChar3, '+');
                        *plusChar = '\0';
                        offsetChar4 = plusChar+1;
                        plusChar = strchr(offsetChar4, '.');
                        *plusChar = '\0';
                        subrow_size = atoi(offsetChar1);
```

```
                        subcol_size = atoi(offsetChar2);
                        subrow_offset = atoi(offsetChar3);
                        subcol_offset = atoi(offsetChar4);
                        WTmessage("subrow_size=%d subcol_size=%d subrow_offset=%d
                                subcol_offset=%d\n", subrow_size, subcol_size, subrow_offset,
                                subcol_offset);

                        /*** convert image boundaries to pan,tilt, fovx, fovy ***/
                        if((subrow_offset+subrow_size)>row_size)
                                subrow_size=row_size-subrow_offset;
                        p1 = pMin + ((float)(subrow_offset)/(float)row_size)*(pMax-pMin);
                        p2 = pMin + ((float)(subrow_offset+subrow_size)/(float)row_size)*
                                (pMax-pMin);
                        if((subcol_offset+subcol_size)>col_size)
                                subcol_size=col_size-subcol_offset;
                        t1 = tMin + ((float)(col_size-subcol_offset-subcol_size)/(float)col_size)*
                                (tMax-tMin);
                        t2 = tMin + ((float)(col_size-subcol_offset)/(float)col_size)*(tMax-tMin);
                        WTmessage("p1=%f p2=%f t1=%f t2=%f\n", degrees(p1), degrees(p2),
                                degrees(t1), degrees(t2));

                        /*** create a 3d image dome model wedge ***/
                        strcpy(path, filepath);
                        strcat(path, WTFILE_DELIM);
                        strcat(path, fname);
                        WTmessage("load image wedge fname: %s\n\n",path);
                        image_planimetric_addM(panCam_planimetric, degrees(p1), degrees(p2),
                                                degrees(t1), degrees(t2), path);
                        WTuniverse_go1();
                    }
                }
            }
            WTdirectory_close(dir);
        WTmessage("\n\n");
        }
}

/*******************************************************************
** 3D overlay function to draw panorama image projections
*******************************************************************/
void calculate_3d_point(float pan, float tilt, WTp3 p)
{
  PanCamInfo *info;
```

```
  info = (PanCamInfo *) WTnode_getdata(panCam_node);

  p[X] = fPanCamProjectionDistance * sin(pan) * cos(tilt);
  p[Y] = fPanCamProjectionDistance * sin(tilt);
  p[Z] = fPanCamProjectionDistance * cos(pan) * cos(tilt);
  WTp3_local2worldframe(p, &info->camera_pose, p);
}

static void panCam_image_projections(WTwindow *w, FLAG eye)
{
	int i;
	WTp3 vect[8], hvect[2], lvect[2], vvect[2];
	int pdiv, tdiv;
	PanCamInfo *info;
	float pMin, pMax, tMin, tMax, tRes, pRes;
	float pIBMin, pIBMax, tIBMin, tIBMax;
	float pan, tilt;
	float xMid, yMid;
	float fovx, fovy;

	info = (PanCamInfo *) WTnode_getdata(panCam_node);
	if(info==NULL) return;

	switch(panCam_type) {
		case HIRES_CAM:
			fovx = HIRES_FOVX; fovy = HIRES_FOVY;
			break;
		case LORES_CAM:
			fovx = LORES_FOVX; fovy = LORES_FOVY;
			break;
	}

	if(info->tRes<1.0 || info->pRes<1.0) return;

	pMin = radians(info->pMin);
	pMax = radians(info->pMax);
	pRes = radians(info->pRes);
	tMin = -radians(info->tMax);
	tMax = -radians(info->tMin);
	tRes = radians(info->tRes);

	WTwindow_set3Dcolor(w, 0, 0, 255);
	if ((pMin > pMax) || (tMin > tMax)) WTwindow_set3Dcolor(w, 255, 0, 0);
```

```
WTp3_copy(info->camera_pose.p, vect[0]);
for (i = 1; i < 4; i++) {
  WTp3_copy(vect[0], vect[2 * i]);
}

pdiv = (int) ((pMax - pMin) / pRes);
pMax = pMin + pRes * pdiv;

tdiv = (int) ((tMax - tMin) / tRes);
tMax = tMin + tRes * tdiv;

/*** Calculate Projection Border Lines ***/
calculate_3d_point(pMax, tMin, vect[1]);
calculate_3d_point(pMax, tMax, vect[3]);
calculate_3d_point(pMin, tMax, vect[5]);
calculate_3d_point(pMin, tMin, vect[7]);
WTwindow_draw3Dlines(w, vect, 8, WTLINE_SEGMENTS);

tilt_count=0;
/*** Calculate Projection Patch Lines ***/
for (tilt = tMin; tilt <= tMax + 0.0001; tilt += tRes) {
  calculate_3d_point(pMin, tilt, lvect[0]);
  /*** count # of images ***/
  tilt_count++;
  pan_count=0;
  for (pan = pMin; pan <= pMax + 0.0001; pan += pRes) {
          calculate_3d_point(pan, tilt, lvect[1]);
          WTwindow_draw3Dlines(w, lvect, 2, WTLINE_SEGMENTS);
          WTp3_copy(lvect[1], lvect[0]);
          pan_count++;
  }
}

for (pan = pMin; pan <= pMax + 0.0001; pan += pRes) {
  calculate_3d_point(pan, tMin, lvect[0]);

  for (tilt = tMin; tilt <= tMax + 0.0001; tilt += tRes) {
    calculate_3d_point(pan, tilt, lvect[1]);
    WTwindow_draw3Dlines(w, lvect, 2, WTLINE_SEGMENTS);
    WTp3_copy(lvect[1], lvect[0]);
  }
}

/*** Calculate Single Image Window Extents ***/
```

```
WTwindow_set3Dcolor(w, 255, 0, 0);
WTwindow_set3Dlinewidth(w, 2);
xMid = (pMax + pMin) / 2;
yMid = (tMax + tMin) / 2;


if (pdiv % 2 == 1) {
pIBMin = xMid - radians(fovx);
pIBMax = xMid;
} else {
        pIBMin = xMid - radians(fovx) / 2;
        pIBMax = xMid + radians(fovx) / 2;
}

if (tdiv % 2 == 1) {
        tIBMin = yMid - radians(fovy);
        tIBMax = yMid;
} else {
        tIBMin = yMid - radians(fovy) / 2;
        tIBMax = yMid + radians(fovy) / 2;
}

/*** Calculate Image Border Lines ***/
calculate_3d_point(pIBMax, tIBMin, vect[0]);
calculate_3d_point(pIBMax, tIBMax, vect[1]);
calculate_3d_point(pIBMin, tIBMax, vect[2]);
calculate_3d_point(pIBMin, tIBMin, vect[3]);
WTwindow_draw3Dlines(w, vect, 4, WTLINE_CLOSE);

/*** Draw in Field of View around Outside Edge of Camera Center Points ***/
pIBMin = pMin - radians(fovx) / 2;
pIBMax = pMax + radians(fovx) / 2;

tIBMin = tMin - radians(fovy) / 2;
tIBMax = tMax + radians(fovy) / 2;

calculate_3d_point(pIBMin, tIBMin, lvect[0]);
calculate_3d_point(pIBMin, tIBMax, hvect[0]);

for (pan = pIBMin; pan <= pIBMax + 0.0001; pan += pRes) {
        calculate_3d_point(pan, tIBMin, lvect[1]);
        WTwindow_draw3Dlines(w, lvect, 2, WTLINE_SEGMENTS);
        WTp3_copy(lvect[1], lvect[0]);
```

```
			calculate_3d_point(pan, tIBMax, hvect[1]);
			WTwindow_draw3Dlines(w, hvect, 2, WTLINE_SEGMENTS);
			WTp3_copy(hvect[1], hvect[0]);
	}

	/*** Closes Pan Display from pRes artifact***/
	calculate_3d_point(pIBMax, tIBMin, lvect[1]);
	WTwindow_draw3Dlines(w, lvect, 2, WTLINE_SEGMENTS);
	calculate_3d_point(pIBMax, tIBMax, hvect[1]);
	WTwindow_draw3Dlines(w, hvect, 2, WTLINE_SEGMENTS);

	calculate_3d_point(pIBMin, tIBMin, lvect[0]);
	calculate_3d_point(pIBMax, tIBMin, hvect[0]);

	for (tilt = tIBMin; tilt <= tIBMax + 0.0001; tilt += tRes) {
			calculate_3d_point(pIBMin, tilt, lvect[1]);
			WTwindow_draw3Dlines(w, lvect, 2, WTLINE_SEGMENTS);
			WTp3_copy(lvect[1], lvect[0]);

			calculate_3d_point(pIBMax, tilt, hvect[1]);
			WTwindow_draw3Dlines(w, hvect, 2, WTLINE_SEGMENTS);
			WTp3_copy(hvect[1], hvect[0]);
	}
	/*** Closes Pan Display from tRes artifact***/
	calculate_3d_point(pIBMin, tIBMax, lvect[1]);
	WTwindow_draw3Dlines(w, lvect, 2, WTLINE_SEGMENTS);
	calculate_3d_point(pIBMax, tIBMax, hvect[1]);
	WTwindow_draw3Dlines(w, hvect, 2, WTLINE_SEGMENTS);

	WTwindow_set3Dlinewidth(w, 1);
}

/*****************************************************************
** 2D overlay function to readout panorama parameters
*****************************************************************/

static void panCam_plan_readout(WTwindow *w, FLAG eye)
{
	int i=1;
	char buf[256];
	PanCamInfo *info;

	info = (PanCamInfo *) WTnode_getdata(panCam_node);
	if(info==NULL) return;
```

```
WTwindow_set2Dfont(w,1);
WTwindow_set2Dcolor(window_id, font_color[0], font_color[1], font_color[0]);
sprintf(buf,"camera ID -- %d (%s)",info->camID,panCamId_text);
WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(7), buf);
sprintf(buf,"resolution 1/%d",info->resolution);
WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(8), buf);
sprintf(buf,"compression %d:1",info->compression);
WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(9), buf);
sprintf(buf,"Pan -- Min:%7.2f Max:%7.2f
        Res:%7.2f",info->pMin,info->pMax,info->pRes);
WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(10), buf);
sprintf(buf,"Tilt -- Min:%7.2f Max:%7.2f Res:%7.2f",info->tMin,info->tMax,info->tRes);

WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(11), buf);

/*** Compute data volume and time statistics ***/
panCam_estTime = pan_count*tilt_count*PANCAM_TIME_PER_PT;
panCam_estDataVol = pan_count*tilt_count*PANCAM_MBITS_PER_IMAGE;
/*** x2 if stereo ***/
if(     info->camID == SCI_CAMERA_HRC_STEREO ||
        info->camID == SCI_CAMERA_HRR_STEREO ||
        info->camID == SCI_CAMERA_HRG_STEREO ||
        info->camID == SCI_CAMERA_HRB_STEREO ||
        info->camID == SCI_CAMERA_WM_STEREO    )
        panCam_estDataVol=2.0f*panCam_estDataVol;
/*** x3 if color ***/
if(     info->camID == SCI_CAMERA_HRC_STEREO ||
        info->camID == SCI_CAMERA_HRC_LEFT ||
        info->camID == SCI_CAMERA_HRC_RIGHT  )
        panCam_estDataVol=2.0f*panCam_estDataVol;
/*** factor in resolution ***/
panCam_estDataVol = panCam_estDataVol/(info->resolution*info->resolution);
/*** factor in compression ***/
panCam_estDataVol = panCam_estDataVol/(info->compression);

sprintf(buf,"#images: %d x %d",pan_count, tilt_count);
if(     info->camID == SCI_CAMERA_HRC_STEREO ||
        info->camID == SCI_CAMERA_HRR_STEREO ||
        info->camID == SCI_CAMERA_HRG_STEREO ||
        info->camID == SCI_CAMERA_HRB_STEREO ||
        info->camID == SCI_CAMERA_WM_STEREO    )  {

        strcat(buf, " x 2");
        sprintf(buf,"%s = %d",buf,pan_count*tilt_count*2);
```

```
	}
	else {
		sprintf(buf,"%s = %d",buf,pan_count*tilt_count);
	}
	WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(12), buf);
	sprintf(buf,"Est. time: %8.2f min.",panCam_estTime);
	WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(13), buf);
	sprintf(buf,"Est. data vol: %f Mbits",panCam_estDataVol);
	WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(14), buf);
}

/*****************************************************************************
*** Pan Cam UI
*****************************************************************************
/
static void BuildPanCamUI(WTui *pwtui)
{
  WTui *pwtuiCamSet, *pwtuiResSet, *pwtuiComSet;
  WTui *pwtuiMenu;
  WTui *pwtuiCamPop, *pwtuiResPop, *pwtuiComPop;
  WTui *Cam;
  WTui *Res;
  WTui *Comp;
  char *PanCamCamSetting = "Stereo Color";
  char *PanCamResSetting = "1/1";
  char *PanCamComSetting = "1:1";
  float fovx, fovy;
  PanCamInfo *info;

  info = (PanCamInfo *) WTnode_getdata(panCam_node);

  pwtuiPanCam = WTuiform_new(pwtui, "Pan Cam",
		WTUIATT_LEFT, (int) (ui_scale * 311.0f), WTUIATT_TOP, 0,
		WTUIATT_WIDTH, (int) (ui_scale * 309.0f), WTUIATT_HEIGHT, (int)
(ui_scale * 275.0f), NULL);

  pwtuiMenu = WTuimenubar_new(pwtuiPanCam);
  pwtuiCamPop = WTuimenupopup_new(pwtuiMenu, "CamID");
  Cam = WTuimenuitem_new(pwtuiCamPop, "Hi-Res, Stereo Color");
  WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
  Cam = WTuimenuitem_new(pwtuiCamPop, "Hi-Res, Stereo Mono (Red)");
  WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
  Cam = WTuimenuitem_new(pwtuiCamPop, "Hi-Res, Stereo Mono (Green)");
  WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
```

```
Cam = WTuimenuitem_new(pwtuiCamPop, "Hi-Res, Stereo Mono (Blue)");
WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
Cam = WTuimenuitem_new(pwtuiCamPop, "Hi-Res, Left Color");
WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
Cam = WTuimenuitem_new(pwtuiCamPop, "Hi-Res, Left Mono (Red)");
WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
Cam = WTuimenuitem_new(pwtuiCamPop, "Hi-Res, Left Mono (Green)");
WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
Cam = WTuimenuitem_new(pwtuiCamPop, "Hi-Res, Left Mono (Blue)");
WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
Cam = WTuimenuitem_new(pwtuiCamPop, "Hi-Res, Right Color");
WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
Cam = WTuimenuitem_new(pwtuiCamPop, "Hi-Res, Right Mono (Red)");
WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
Cam = WTuimenuitem_new(pwtuiCamPop, "Hi-Res, Right Mono (Green)");
WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
Cam = WTuimenuitem_new(pwtuiCamPop, "Hi-Res, Right Mono (Blue)");
WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
Cam = WTuimenuitem_new(pwtuiCamPop, "Lo-Res, Stereo");
WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
Cam = WTuimenuitem_new(pwtuiCamPop, "Lo-Res, Left");
WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);
Cam = WTuimenuitem_new(pwtuiCamPop, "Lo-Res, Right");
WTui_setcallback(Cam, WTUIEVENT_ACTIVATE, PanCamSelectCamera, NULL);


pwtuiResPop = WTuimenupopup_new(pwtuiMenu, "Res");
Res = WTuimenuitem_new(pwtuiResPop, "1/1");
WTui_setcallback(Res, WTUIEVENT_ACTIVATE, PanCamSelectRes, NULL);
Res = WTuimenuitem_new(pwtuiResPop, "1/2");
WTui_setcallback(Res, WTUIEVENT_ACTIVATE, PanCamSelectRes, NULL);
Res = WTuimenuitem_new(pwtuiResPop, "1/3");
WTui_setcallback(Res, WTUIEVENT_ACTIVATE, PanCamSelectRes, NULL);
Res = WTuimenuitem_new(pwtuiResPop, "1/4");
WTui_setcallback(Res, WTUIEVENT_ACTIVATE, PanCamSelectRes, NULL);

pwtuiComPop = WTuimenupopup_new(pwtuiMenu, "Comp");
Comp = WTuimenuitem_new(pwtuiComPop, "1:1");
WTui_setcallback(Comp, WTUIEVENT_ACTIVATE, PanCamSelectComp, NULL);
Comp = WTuimenuitem_new(pwtuiComPop, "8:1");
WTui_setcallback(Comp, WTUIEVENT_ACTIVATE, PanCamSelectComp, NULL);

Comp = WTuimenuitem_new(pwtuiComPop, "16:1");
WTui_setcallback(Comp, WTUIEVENT_ACTIVATE, PanCamSelectComp, NULL);
```

```
Comp = WTuimenuitem_new(pwtuiComPop, "32:1");
WTui_setcallback(Comp, WTUIEVENT_ACTIVATE, PanCamSelectComp, NULL);

fovx = HIRES_FOVX; fovy = HIRES_FOVY;
pwtuiPanResSlider = WTuiscale_new(pwtuiPanCam, "Pan Res", 10.0f*fovx / 2.2, 10.0f*fovx,
        1, 10.0f*info->pRes,
        WTUIATT_LEFT, (int) (ui_scale * 5.0f), WTUIATT_TOP, (int) (ui_scale * 50.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 140.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);

pwtuiTiltResSlider = WTuiscale_new(pwtuiPanCam, "Tilt Res", 10.0f*fovy / 2.2, 10.0f*fovy, 1,
        10.0f*info->tRes,
        WTUIATT_LEFT, (int) (ui_scale * 165.0f), WTUIATT_TOP, (int) (ui_scale * 50.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 140.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);

pwtuiPanMinSlider = WTuiscale_new(pwtuiPanCam, "Pan Min", PAN_MIN, PAN_MAX, 0,
        info->pMin,
        WTUIATT_LEFT, (int) (ui_scale * 5.0f), WTUIATT_TOP, (int) (ui_scale * 115.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 140.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);

pwtuiTiltMinSlider = WTuiscale_new(pwtuiPanCam, "Tilt Min", TILT_MIN, TILT_MAX, 0,
        info->tMin,
        WTUIATT_LEFT, (int) (ui_scale * 165.0f), WTUIATT_TOP, (int) (ui_scale * 115.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 140.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);

pwtuiPanMaxSlider = WTuiscale_new(pwtuiPanCam, "Pan Max", PAN_MIN, PAN_MAX, 0,
        info->pMax,
        WTUIATT_LEFT, (int) (ui_scale * 5.0f), WTUIATT_TOP, (int) (ui_scale * 180.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 140.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);

pwtuiTiltMaxSlider = WTuiscale_new(pwtuiPanCam, "Tilt Max", TILT_MIN, TILT_MAX, 0,
        info->tMax,
        WTUIATT_LEFT, (int) (ui_scale * 165.0f), WTUIATT_TOP, (int) (ui_scale * 180.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 140.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);

WTui_setcallback(pwtuiPanResSlider, WTUIEVENT_ACTIVATE, PanCamPanResSlider,
        NULL);
WTui_setcallback(pwtuiTiltResSlider, WTUIEVENT_ACTIVATE, PanCamTiltResSlider,
        NULL);
```

```
  WTui_setcallback(pwtuiPanMinSlider, WTUIEVENT_ACTIVATE, PanCamPanMinSlider,
          NULL);
  WTui_setcallback(pwtuiPanMaxSlider, WTUIEVENT_ACTIVATE, PanCamPanMaxSlider,
          NULL);
  WTui_setcallback(pwtuiTiltMinSlider, WTUIEVENT_ACTIVATE, PanCamTiltMinSlider,
          NULL);
  WTui_setcallback(pwtuiTiltMaxSlider, WTUIEVENT_ACTIVATE, PanCamTiltMaxSlider,
          NULL);

  WTui_manage(pwtuiPanCam);
}

static void PanCamSelectCamera(WTui *pStruct, void *pData)
{
  PanCamInfo *info;
  int old_panCamType;
  char *camId_text;

  old_panCamType = panCam_type;
  camId_text = WTui_gettext(pStruct);
  sprintf(panCamId_text,"%s",camId_text);
  info = (PanCamInfo *) WTnode_getdata(panCam_node);
  if(!strcmp(panCamId_text,"Hi-Res, Stereo Color"))
          {info->camID = SCI_CAMERA_HRC_STEREO;  panCam_type=HIRES_CAM;}
  else if(!strcmp(panCamId_text,"Hi-Res, Stereo Mono (Red)"))
          {info->camID = SCI_CAMERA_HRR_STEREO;  panCam_type=HIRES_CAM;}
  else if(!strcmp(panCamId_text,"Hi-Res, Stereo Mono (Green)"))
          {info->camID = SCI_CAMERA_HRG_STEREO;  panCam_type=HIRES_CAM;}
  else if(!strcmp(panCamId_text,"Hi-Res, Stereo Mono (Blue)"))
          {info->camID = SCI_CAMERA_HRB_STEREO;  panCam_type=HIRES_CAM;}
  else if(!strcmp(panCamId_text,"Hi-Res, Left Color"))
          {info->camID = SCI_CAMERA_HRC_LEFT;    panCam_type=HIRES_CAM;}
  else if(!strcmp(panCamId_text,"Hi-Res, Left Mono (Red)"))
          {info->camID = SCI_CAMERA_HRR_LEFT;    panCam_type=HIRES_CAM;}
  else if(!strcmp(panCamId_text,"Hi-Res, Left Mono (Green)"))
          {info->camID = SCI_CAMERA_HRG_LEFT;    panCam_type=HIRES_CAM;}
  else if(!strcmp(panCamId_text,"Hi-Res, Left Mono (Blue)"))
          {info->camID = SCI_CAMERA_HRB_LEFT;    panCam_type=HIRES_CAM;}
  else if(!strcmp(panCamId_text,"Hi-Res, Right Color"))
          {info->camID = SCI_CAMERA_HRC_RIGHT;   panCam_type=HIRES_CAM;}
  else if(!strcmp(panCamId_text,"Hi-Res, Right Mono (Red)"))
          {info->camID = SCI_CAMERA_HRR_RIGHT;   panCam_type=HIRES_CAM;}
  else if(!strcmp(panCamId_text,"Hi-Res, Right Mono (Green)"))
          {info->camID = SCI_CAMERA_HRG_RIGHT;   panCam_type=HIRES_CAM;}
```

```
 else if(!strcmp(panCamId_text,"Hi-Res, Right Mono (Blue)"))
        {info->camID = SCI_CAMERA_HRB_RIGHT;   panCam_type=HIRES_CAM;}
 else if(!strcmp(panCamId_text,"Lo-Res, Stereo"))
        {info->camID = SCI_CAMERA_WM_STEREO;   panCam_type=LORES_CAM;}
 else if(!strcmp(panCamId_text,"Lo-Res, Left"))
        {info->camID = SCI_CAMERA_WM_LEFT;  panCam_type=LORES_CAM;}
 else if(!strcmp(panCamId_text,"Lo-Res, Right"))
        {info->camID = SCI_CAMERA_WM_RIGHT;    panCam_type=LORES_CAM;}
 WTmessage("Camera '%s' selected\n",panCamId_text);
 WTmessage("CamID = %d\n",info->camID);
 if(panCam_type != old_panCamType) PanCamResetSliders(panCam_type);
}

static void PanCamSelectRes(WTui *pStruct, void *pData)
{
 PanCamInfo *info;
 char *res_text;

 res_text = WTui_gettext(pStruct);
 info = (PanCamInfo *) WTnode_getdata(panCam_node);
 if(!strcmp(res_text,"1/1"))             {info->resolution=1;}
 else if(!strcmp(res_text,"1/2"))        {info->resolution=2;}
 else if(!strcmp(res_text,"1/3"))        {info->resolution=3;}
 else if(!strcmp(res_text,"1/4"))        {info->resolution=4;}
 WTmessage("Resolution = %s (%d)\n",res_text, info->resolution);
}

static void PanCamSelectComp(WTui *pStruct, void *pData)
{
 PanCamInfo *info;
 char *comp_text;

 comp_text = WTui_gettext(pStruct);
 info = (PanCamInfo *) WTnode_getdata(panCam_node);
 if(!strcmp(comp_text,"1:1"))                    {info->compression=1;}
 else if(!strcmp(comp_text,"8:1"))     {info->compression=8;}
 else if(!strcmp(comp_text,"16:1"))   {info->compression=16;}
 else if(!strcmp(comp_text,"32:1"))   {info->compression=32;}
 WTmessage("Compression = %s (%d)\n",comp_text, info->compression);
}

static void PanCamResetSliders(int panCamType)
{
        float fovx, fovy;
```

```
        float panRes_max, panRes_min;
        float tiltRes_max, tiltRes_min;
        PanCamInfo *info;

        info = (PanCamInfo *) WTnode_getdata(panCam_node);

        WTui_delete(pwtuiPanResSlider);
        WTui_delete(pwtuiTiltResSlider);
        switch(panCamType) {
                case HIRES_CAM:
                        fovx = HIRES_FOVX; fovy = HIRES_FOVY;
                        info->pRes=10; info->tRes=8;
                        break;
                case LORES_CAM:
                        fovx = LORES_FOVX; fovy = LORES_FOVY;
                        info->pRes=31.5; info->tRes=24.0;
                        break;
        }
        panRes_min=10.0f*fovx / 2.2;
        panRes_max=10.0f*fovx;
        tiltRes_min=10.0f*fovy / 2.2;
        tiltRes_max=10.0f*fovy;
        pwtuiPanResSlider = WTuiscale_new(pwtuiPanCam, "Pan Res", panRes_min,
                panRes_max, 1, 10.0f*info->pRes,
                WTUIATT_LEFT, (int) (ui_scale * 5.0f), WTUIATT_TOP, (int) (ui_scale *
                50.0f), WTUIATT_WIDTH, (int) (ui_scale * 140.0f), WTUIATT_HEIGHT, (int)
                (ui_scale * 20.0f), NULL);
          pwtuiTiltResSlider = WTuiscale_new(pwtuiPanCam, "Tilt Res", tiltRes_min,
                tiltRes_max, 1, 10.0f*info->tRes, WTUIATT_LEFT, (int) (ui_scale * 165.0f),
                WTUIATT_TOP, (int) (ui_scale * 50.0f), WTUIATT_WIDTH, (int) (ui_scale *
                140.0f), WTUIATT_HEIGHT, (int) (ui_scale * 20.0f), NULL);
        WTui_setcallback(pwtuiPanResSlider, WTUIEVENT_ACTIVATE,
                PanCamPanResSlider, NULL);
        WTui_setcallback(pwtuiTiltResSlider, WTUIEVENT_ACTIVATE,
                PanCamTiltResSlider, NULL);
}

static void PanCamPanResSlider(WTui *pStruct, void *pData)
{
  float *fPCM;
  PanCamInfo *info;

  info = (PanCamInfo *) WTnode_getdata(panCam_node);
```

```
  fPCM = (float *) pData;

  info->pRes = *fPCM;
}

static void PanCamTiltResSlider(WTui *pStruct, void *pData)
{
  float *fPCM;
  PanCamInfo *info;

  info = (PanCamInfo *) WTnode_getdata(panCam_node);

  fPCM = (float *) pData;

  info->tRes = *fPCM;
}

static void PanCamPanMinSlider(WTui *pStruct, void *pData)
{
  float *fPCM;
  PanCamInfo *info;

  info = (PanCamInfo *) WTnode_getdata(panCam_node);

  fPCM = (float *) pData;

  info->pMin = *fPCM;
}

static void PanCamPanMaxSlider(WTui *pStruct, void *pData)
{
  float *fPCM;
  PanCamInfo *info;

  info = (PanCamInfo *) WTnode_getdata(panCam_node);

  fPCM = (float *) pData;

  info->pMax = *fPCM;
}

static void PanCamTiltMinSlider(WTui *pStruct, void *pData)
{
  float *fPCM;
```

```
  PanCamInfo *info;

  info = (PanCamInfo *) WTnode_getdata(panCam_node);

  fPCM = (float *) pData;

  info->tMin = *fPCM;
}

static void PanCamTiltMaxSlider(WTui *pStruct, void *pData)
{
  float *fPCM;
  PanCamInfo *info;

  info = (PanCamInfo *) WTnode_getdata(panCam_node);

  fPCM = (float *) pData;

  info->tMax = *fPCM;
}
```

Appendix C

Spectrometer Planner Source Code

This is a complete listing of the source code (in C) for the spectrometer planner. After the primary implementation by Allport, additions were made by Ted Blackmon. Blackmon's changes are included.

C.1 Listing of spectral.h

```
/***************************************************************************
**
** spectral.h - planning module for bore-sighted spectrometer on Marsokhod rover
**
****************************************************************************/
void spectral_init();
void spectral_loop();
void spectral_exit();
WTnode *spectral_newTask(WTnode *);
void spectral_editTask(WTnode *);
void spectral_seq(WTnode *node, FILE *file);
```

C.2 Listing of spectral.c

```
/***************************************************************************
** spectral.c - planning module for bore-sighted spectrometer on Marsokhod rover
**
** Written by Christopher S. Allport & Theodore T. Blackmon
** Copyright 1999.
****************************************************************************/

#include <stdio.h>
```

```
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <assert.h>
#include <libgen.h>

#include "wt.h"
#include "ims.h"

#include "cursor3d.h"
#include "inputs.h"
#include "overlay.h"
#include "text3d.h"

#include "rover_plan.h"
#include "marsokhod.h"

/**********************************************************/
void spectral_init();
void spectral_loop();
void spectral_exit();
WTnode *spectral_newTask(WTnode *marso);
void spectral_editTask(WTnode *node);
void spectral_seq(WTnode *node, FILE *file);

/**********************************************************/

typedef struct {
  int spectrometerID;
  int science_site;
  WTpq featurePose;
  char *featureImage;
  float featurePx;
  float featurePy;
  WTpq spectrometerPose;
  float pan;
  float tilt;
  int pSteps, tSteps;
  float pInc, tInc;
  int camID;
  int returnSpectra;
  int testCarbonate;
  int ic_spectraDataFlag;
  int ic_hiResImageFlag;
```

```
  int ic_traverseFlag;
  int nc_spectraDataFlag;
  int nc_hiResImageFlag;
  int nc_traverseFlag;
  int tn_reAcquireFlag;
  int tn_spectraDataFlag;
  int tn_hiResImageFlag;
  int tn_traverseFlag;

} SpectralInfo;

#define NEAR_IR                 0
#define MID_IR                  1


/*** spectral nodes ***/
static WTnode *spectral_taskNode;
static WTnode *spectral_node;

/*** spectral state machine ***/
static int spectral_state;
static int last_spectral_state;

#define SPECTRAL_IDLE                   0
#define SPECTRAL_SELECT_FEATURE                 1
#define SPECTRAL_ANCHOR_FEATURE                         2

/*** Default Defines ***/
#define DEFAULT_PAN                     0
#define DEFAULT_TILT                    0
#define DEFAULT_SPEC_ID                         NEAR_IR
#define DEFAULT_PSTEPS                  1
#define DEFAULT_PINC                    1.0f
#define DEFAULT_TSTEPS                  1
#define DEFAULT_TINC                    1.0f

#define SPEC_FOV                1
#define SPECT_FOV               1
#define HOT_SPOT_TESS                   16
#define SPECT_TESS                      16

static void spectral_idle_entry();
static void spectral_idle_loop();
static void spectral_idle_exit();
```

```
static void spectral_selectFeature_entry();
static void spectral_selectFeature_loop();
static void spectral_selectFeature_exit();


/*** spectral UI ***/
static void BuildSpectralUI(WTui *pwtui);

WTui *pwtuiIdPop;
WTui *pwtuiExpPop;
WTui *pwtuiSelectPop;

static void SpectralNearIR(WTui *pStruct, void *pData);
static void SpectralMidIR(WTui *pStruct, void *pData);
static void SpectralCube(WTui *pStruct, void *pData);
static void SpectralFeature(WTui *pStruct, void *pData);
static void SpectralParams(WTui *pStruct, void *pData);

static void ParamsAlwaysReturn(WTui *pStruct, void *pData);
static void ParamsTestBit(WTui *pStruct, void *pData);

static void ParamsICCompare(WTui *pStruct, void *pData);
static void ParamsNCCompare(WTui *pStruct, void *pData);
static void ParamsTNCompare(WTui *pStruct, void *pData);

static void ParamsDoCapture(WTui *pStruct, void *pData);
static void ParamsCameraSelect(WTui *pStruct, void *pData);
static void SetParamsClose(WTui *pStruct, void *pData);

static WTui *texture_form;
static WTwindow *texture_window;
static WTnode *texture_root=NULL;
static char *texture_buf=NULL;
static WTui *pwtuiShell;

float calculate_3d_point(WTp3 wtp3Origin, WTp3 wtp3Dir, float fDist, WTp3 wtp3Point, FLAG
        delta);

static void spectral_projections(WTwindow *w, FLAG eye);
static void spectral_featureTexture_display(WTwindow *w, FLAG eye);
static void spectral_plan_readout(WTwindow *w, FLAG eye);

/****************************************************************
**  Initialization function for pan cam planning
```

```
*****************************************************************/

void spectral_init()
{
	WTui *ui;

	WTmessage("Spectral Init Function Called\n");
	/*** activate rover planning tool ***/
	tool_activate(roverPlan_control, TRUE);

	/*** Build planning UI for Pan Cam Imager ***/
	ui = BuildGeneralUI();
	BuildSpectralUI(ui);
	WTui_manage(ui);

	/*** default interactivity is idle ***/
	spectral_state = SPECTRAL_IDLE;

	/*** add 3d overlay for panorama image projection window ***/
	overlay3d_add(spectral_projections);
	/*** add 2d overlay function to read out spectral experiment parameters ***/
	overlay2d_add(spectral_plan_readout);
}

void spectral_loop()
{
  if(spectral_state != last_spectral_state)  {
	switch(last_spectral_state)  {
		/* exit functions */
		case SPECTRAL_IDLE:
			printf(">> Exiting spectral IDLE mode ...\n");
			spectral_idle_exit();
			printf("done.\n\n");
			break;
		case SPECTRAL_SELECT_FEATURE:
			printf(">> Exiting spectral SELECT_FEATURE mode ...\n");
			spectral_selectFeature_exit();
			printf("done.\n\n");
			break;
	}
	switch(spectral_state)  {
		/* entry functions */
		case SPECTRAL_IDLE:
			printf(">> Entering spectral IDLE mode ...\n");
```

```
                spectral_idle_entry();
                printf("done.\n\n");
                break;
            case SPECTRAL_SELECT_FEATURE:
                printf(">> Entering spectral SELECT_FEATURE mode ...\n");
                spectral_selectFeature_entry();
                printf("done.\n\n");
                break;
        }
        last_spectral_state = spectral_state;
    }
    switch(spectral_state)  {

            case SPECTRAL_IDLE:
                spectral_idle_loop();
                break;
            case SPECTRAL_SELECT_FEATURE:
                spectral_selectFeature_loop();
                break;
    }
}

void spectral_exit()
{
        /*** remove 3d overlay for spectral experiment projection window ***/
        overlay3d_delete(spectral_projections);
        /*** remove 2d overlay function to read out spectral experiment parameters ***/
        overlay2d_delete(spectral_plan_readout);
        /*** delete texture form window ***/
        WTnode_delete(texture_root);
        WTwindow_delete(texture_window);
        WTmessage("Leaving Spectral\n");
}

WTnode *spectral_newTask(WTnode *marso)
{
        WTnode *xform;
        SpectralInfo *info;
        WTpq pose1, pose2;
        WTp3 pos;

        WTmessage("Create a new pan-cam node!\n");

        /*** make a sep, xform, and geom node ***/
```

```
spectral_taskNode = WTsepnode_new(NULL);
xform = WTxformnode_new(spectral_taskNode);
pos[X] = 0.0f;
pos[Y] = 0.0f;
pos[Z] = -1.6f;
WTnode_settranslation(xform, pos);
spectral_node = WTgeometrynode_new(spectral_taskNode,
                    WTgeometry_newsphere(0.5, 8, 8, FALSE, TRUE));
WTnode_enable(spectral_node, FALSE);

/*** allocate memory for pan-cam info struct ***/
info = (SpectralInfo *)malloc(sizeof(SpectralInfo));
if(info == NULL) printf("malloc failed for spectralInfo struct.\n");

/*** Set up default values ***/
info->featureImage = NULL;
info->featurePx=-1.0f;
info->featurePy=-1.0f;
info->pan = DEFAULT_PAN;
info->tilt = DEFAULT_TILT;
info->spectrometerID = DEFAULT_SPEC_ID;
info->pSteps=DEFAULT_PSTEPS;
info->pInc=DEFAULT_PINC;
info->tSteps=DEFAULT_TSTEPS;
info->tInc=DEFAULT_TINC;

info->camID = 0;
info->returnSpectra = TRUE;
info->testCarbonate = FALSE;
info->ic_spectraDataFlag = TRUE;
info->ic_hiResImageFlag = FALSE;
info->ic_traverseFlag = FALSE;
info->nc_spectraDataFlag = TRUE;
info->nc_hiResImageFlag = FALSE;
info->nc_traverseFlag = FALSE;
info->tn_reAcquireFlag = FALSE;
info->tn_spectraDataFlag = TRUE;
info->tn_hiResImageFlag = FALSE;
info->tn_traverseFlag = FALSE;

/*** Set Rover Camera Position ***/
marsokhod_panCam_xform(WTnode_getchild(marso,0), &info->spectrometerPose);
WTpq_init(&pose1); WTpq_init(&pose2);
pos[X] = 0.05f;
```

```
        pos[Y] = 0.01f;
        pos[Z] = -0.125f;
        WTq_copy(info->spectrometerPose.q, pose2.q);
        WTp3_world2localframe(pos, &pose2, pose1.p);
        info->spectrometerPose.p[X] += pose1.p[Z];
        info->spectrometerPose.p[Y] -= pose1.p[Y];
        info->spectrometerPose.p[Z] += pose1.p[X];

        /*** set default value of pan-cam info struct ***/
        WTnode_setdata(spectral_node, (void *)info);

        return(spectral_taskNode);
}

/*********************************************************************
** Spectral task edit function
*********************************************************************/
void spectral_editTask(WTnode *node)
{
        WTmessage("Edit spectral node!\n");
        spectral_node = WTnode_getchild(node, 1);
}

/*********************************************************************
** Spectral sequence generation function
*********************************************************************/
void spectral_seq(WTnode *node, FILE *file)
{
        SpectralInfo *info;
        spectral_node = WTnode_getchild(node, 1);
        info = (SpectralInfo *) WTnode_getdata(spectral_node);

        fprintf(file, "      :featureimage \"%s\"\n",info->featureImage);
        fprintf(file, "      :featurepx %f\n",info->featurePx);
        fprintf(file, "      :featurepy %f\n",info->featurePy);
        fprintf(file, "      ;for spectral\n");
        fprintf(file, "      :pan1 %f\n",info->pan);
        fprintf(file, "      :tilt1 %f\n",info->tilt);
        fprintf(file, "      :pSteps %d\n",info->pSteps);
        fprintf(file, "      :pInc %f\n",radians(info->pInc));
        fprintf(file, "      :tSteps %d\n",info->tSteps);
        fprintf(file, "      :tInc %f\n",radians(info->tInc));
        fprintf(file, "      :camid_list (%d)\n",info->camID);
        fprintf(file, "      :returnspectra %d0\n", info->returnSpectra);
```

```
	fprintf(file, "      :testcarbonate %d\n", info->testCarbonate);
	fprintf(file, "      :ic_spectradataflag %d\n", info->ic_spectraDataFlag);
	fprintf(file, "      :ic_hiresimageflag %d\n", info->ic_hiResImageFlag);
	fprintf(file, "      :ic_traverseflag %d\n", info->ic_traverseFlag);
	fprintf(file, "      :nc_spectradataflag %d\n", info->nc_spectraDataFlag);
	fprintf(file, "      :nc_hiresimageflag %d\n", info->nc_hiResImageFlag);
	fprintf(file, "      :nc_traverseflag %d\n", info->nc_traverseFlag);
	fprintf(file, "      :tn_reacquireflag %d\n", info->tn_reAcquireFlag);
	fprintf(file, "      :tn_spectradataflag %d\n", info->tn_spectraDataFlag);
	fprintf(file, "      :tn_hiresimageflag %d\n", info->tn_hiResImageFlag);
	fprintf(file, "      :tn_traverseflag %d\n", info->tn_traverseFlag);
	fprintf(file, "      :basefilename ""\n");
}

/****************************
****   Spectral IDLE mode
****************************/
static void spectral_idle_entry()
{
}

static void spectral_idle_loop()
{
}

static void spectral_idle_exit()
{
}

/*********************************
****   Spectral SELECT_FEATURE mode
*********************************/
static void spectral_selectFeature_entry()
{
//	overlay2d_add(info_readout);
	/*** set cursor style to invisible ***/
	cursor3d_style_set(INVISIBLE);
	cursor3d_add();
}

static void spectral_selectFeature_loop()
{
	int i;
	WTpq pose;
```

```
WTnode *info_node;
WTnodepath *npath;
char *node_name;
char *texname;
float texture_u, texture_v;
WTtextureinfo texinfo;
SpectralInfo *info;
WTp3 p[3];
float uTex[3], vTex[3];
float d1,d2,d3,d12,d13,s1,s2;
float uMin,uMax,vMin,vMax;
int iuMin,iuMax,ivMin,ivMax;

cursor3d_update_pos();
if(mouse_data.mbutton) {
        spectral_state = SPECTRAL_ANCHOR_FEATURE;
}
for(i=WTnodepath_numnodes(poly_path)-1; i>0; i--)  {
   info_node = WTnodepath_getnode(poly_path,i);
   if(WTnode_gettype(info_node) == WTNODE_GEOM)  {
                npath = WTnodepath_new(info_node, ims_root, 0);
                WTnodepath_gettranslation(npath, pose.p);
                WTnodepath_getorientation(npath, pose.q);
   }
   if(WTnode_gettype(info_node) == WTNODE_SEP)  {
        node_name = WTnode_getname(info_node);
        info = (SpectralInfo *) WTnode_getdata(spectral_node);
        WTpq_copy(&cursor3d.pose, &info->featurePose);
        if (WTpoly_gettextureinfo(poly_intersected, &texinfo)) {
                WTpoly_getuv(poly_intersected, uTex, vTex);
                uMin = uMax = uTex[0]; vMin = vMax = vTex[0];
                iuMin=0; iuMax=0; ivMin=0; ivMax=0;
                for(i=1; i<3; i++)  {
                        if(uTex[i]<uMin)  {uMin=uTex[i]; iuMin=i;}
                        if(uTex[i]>uMax)  {uMax=uTex[i]; iuMax=i;}
                        if(vTex[i]<vMin)  {vMin=vTex[i]; ivMin=i;}
                        if(vTex[i]>vMax)  {vMax=vTex[i]; ivMax=i;}
                }
                WTgeometry_getvertexposition(WTpoly_getgeometry(poly_intersected),
                                WTpoly_getvertex(poly_intersected,0), p[0]);
                WTgeometry_getvertexposition(WTpoly_getgeometry(poly_intersected),
                                WTpoly_getvertex(poly_intersected,1), p[1]);
                WTgeometry_getvertexposition(WTpoly_getgeometry(poly_intersected),
                                WTpoly_getvertex(poly_intersected,2), p[2]);
```

```
                        WTp3_local2worldframe(p[0], &pose, p[0]);
                        WTp3_local2worldframe(p[1], &pose, p[1]);
                        WTp3_local2worldframe(p[2], &pose, p[2]);
                        d1 = WTp3_distance(p[iuMin],cursor3d.pose.p);
                        d2 = WTp3_distance(p[iuMax],cursor3d.pose.p);
                        d12 = WTp3_distance(p[iuMin],p[iuMax]);
                        s1 = (d1*d1 + d12*d12 - d2*d2)/(2.0f*d12*d12);
                        texture_u = uMin + s1*(uMax-uMin);
                        d1 = WTp3_distance(p[ivMin],cursor3d.pose.p);
                        d2 = WTp3_distance(p[ivMax],cursor3d.pose.p);
                        d12 = WTp3_distance(p[ivMin],p[ivMax]);
                        s1 = (d1*d1 + d12*d12 - d2*d2)/(2.0f*d12*d12);
                        texture_v = vMin + s1*(vMax-vMin);
                        texture_buf = texinfo.name;
                        info->featurePx = texture_u;
                        info->featurePy = texture_v;
                        info->featureImage = basename(texinfo.name);
                    }
                    return;
                }
            }
}

static void spectral_selectFeature_exit()
{
//      overlay2d_delete(info_readout);
        /*** reset cursor style to xhair ***/
        cursor3d_style_set(XHAIR);
        cursor3d_remove();
}

/*********************************
****   Spectral ANCHOR FEATURE mode
*********************************/
static void spectral_anchorFeature_entry()
{
}

static void spectral_anchorFeature_loop()
{
}

static void spectral_anchorFeature_exit()
{
```

```
}

/*****************************************************************
** 3D overlay function to draw spectral experiment projections
*****************************************************************/
float calculate_3d_point(WTp3 wtp3Origin, WTp3 wtp3Dir, float fDist, WTp3 wtp3Point, FLAG
        delta)
{
        float fLen;

        if(WTnode_rayintersect(ims_root, wtp3Dir, wtp3Origin, &fLen, NULL) == NULL)
                fLen = fDist;

/*** This makes the circle appear on the surface - linewidth doesn't work on WTKDirect ***/
        if (delta) fLen -= 0.005f;

        WTp3_mults(wtp3Dir, fLen);
        WTp3_add(wtp3Dir, wtp3Origin, wtp3Point);

        return fLen;
}

void calculate_circle_point(float dx, float dy, float dist, WTp3 p, WTp3 origin)
{
  SpectralInfo *info;
  float fDistance;
  WTp3 dir, dirSpec;

  dirSpec[X] = sin(dx) * cos(dy);
  dirSpec[Y] = sin(dy);
  dirSpec[Z] = cos(dx) * cos(dy);

  info = (SpectralInfo *) WTnode_getdata(spectral_node);

  WTp3_local2worldframe(dirSpec, &info->spectrometerPose, dir);

  fDistance = dist/cos(radians(0.5f*SPEC_FOV));
  p[X] = dirSpec[X]*0.975f*fDistance;
  p[Y] = dirSpec[Y]*0.975f*fDistance;
  p[Z] = dirSpec[Z]*0.975f*fDistance;

  WTp3_local2worldframe(p, &info->spectrometerPose, p);
}
```

```
static void spectral_projections(WTwindow *win, FLAG eye)
{
  int i;
  float dx, dy, fLen;
  WTp3 wtp3Origin, wtp3Dir, wtp3Temp;
  WTq wtqOrigin, wtqQ;
  WTp3 wtp3Vect[SPECT_TESS];
  SpectralInfo *info;

  WTwindow_set3Dcolor(win, 0, 255, 0);
  WTwindow_set3Dlinewidth(win, 2.0f);

  info = (SpectralInfo *) WTnode_getdata(spectral_node);

  WTp3_copy(info->spectrometerPose.p, wtp3Origin);
  WTp3_copy(info->featurePose.p, wtp3Temp);
  WTp3_subtract(wtp3Temp, wtp3Origin, wtp3Temp);
  WTp3_norm(wtp3Temp);

  WTeuler_2q(wtp3Temp[X], wtp3Temp[Y], wtp3Temp[Z], wtqOrigin);

  WTp3_copy(wtp3Origin, wtp3Vect[0]);

  fLen = calculate_3d_point(wtp3Origin, wtp3Temp, 50.0f, wtp3Vect[1], FALSE);
  WTwindow_draw3Dlines(win, wtp3Vect, 2, WTLINE_SEGMENTS);

  for (i = 0; i < SPECT_TESS; i++) {

    dx = (float) cos(radians(i * 360.0f / SPECT_TESS)) * SPECT_FOV / 2;
    dy = (float) sin(radians(i * 360.0f / SPECT_TESS)) * SPECT_FOV / 2;

    WTeuler_2q((float) radians(dx), (float) radians(dy), 0.0f, wtqQ);

    WTp3_rotate(wtp3Temp, wtqQ, wtp3Dir);

    calculate_3d_point(wtp3Origin, wtp3Dir, fLen, wtp3Vect[i], TRUE);
  }

  WTwindow_set3Dlinewidth(win, 3.0f);
  WTwindow_draw3Dlines(win, wtp3Vect, SPECT_TESS, WTLINE_CLOSE);
  WTwindow_set3Dlinewidth(win, 1.0f);
}
```

```
/******************************************************************
** 2D overlay function to draw texture with spectral feature
******************************************************************/

static void spectral_featureTexture_display(WTwindow *w, FLAG eye)
{
	WTp2 xy[4];
	WTp2 uv[4];
	char buf[256];
	SpectralInfo *info;

	info = (SpectralInfo *) WTnode_getdata(spectral_node);

	/** set image and screen coordinates for selected texture **/
	xy[0][X] = 0.0f; xy[0][Y] = 0.0f; uv[0][X] = 0.0f; uv[0][Y] = 0.0f;
	xy[1][X] = 1.0f; xy[1][Y] = 0.0f; uv[1][X] = 1.0f; uv[1][Y] = 0.0f;
	xy[2][X] = 1.0f; xy[2][Y] = 1.0f; uv[2][X] = 1.0f; uv[2][Y] = 1.0f;
	xy[3][X] = 0.0f; xy[3][Y] = 1.0f; uv[3][X] = 0.0f; uv[3][Y] = 1.0f;

	/** draw 2d texture in specified sub window **/
	if(texture_buf!=NULL)  {
		WTwindow_draw2Dtexture(w, texture_buf, FALSE, xy, uv);

	}

	/** draw spectral feature coordinates in image ***/
	WTwindow_set2Dcolor(w,0,255,0);
	WTwindow_draw2Dline(w, info->featurePx-0.1f, info->featurePy, info->featurePx+0.1f,
		info->featurePy);
	WTwindow_draw2Dline(w, info->featurePx, info->featurePy-0.1f, info->featurePx,
		info->featurePy+0.1f);

	/*** draw 2d text overlay for texture name  ***/
	WTwindow_set2Dcolor(w,0,0,255);
	WTwindow_set2Dfont(w,1);
	WTwindow_draw2Dtext(w, 0.10, 0.10, info->featureImage);
}

/******************************************************************
** 2D overlay function to readout panorama parameters
******************************************************************/
static void spectral_plan_readout(WTwindow *w, FLAG eye)
{
	int i=1;
```

```
	char buf0[256];
	char buf[256];
	SpectralInfo *info;

	info = (SpectralInfo *) WTnode_getdata(spectral_node);
	WTwindow_set2Dfont(w,1);
	WTwindow_set2Dcolor(window_id, font_color[0], font_color[1], font_color[0]);

	if(info->spectrometerID==NEAR_IR) sprintf(buf0, "Near IR");
	else if(info->spectrometerID==MID_IR) sprintf(buf0, "Mid IR");
	sprintf(buf,"spectrometer ID -- %d (%s)",info->spectrometerID,buf0);
	WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(7), buf);
	sprintf(buf,"Image with target -- %s",info->featureImage);
	WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(8), buf);
	sprintf(buf,"Image coords (u,v) -- (%3.2f %3.2f)",info->featurePx, info->featurePy);

	WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(9), buf);
	sprintf(buf,"Pointing coords (deg) -- (%7.2f,%7.2f)",degrees(info->pan),
		degrees(info->tilt));
	WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(10), buf);
	sprintf(buf,"Target pos (m) -- (%7.2f,%7.2f,%7.2f)",
			info->featurePose.p[X],info->featurePose.p[Y],info->featurePose.p[Z]);

	WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(11), buf);
	sprintf(buf,"Target range (m) -- (%7.3f)",
			WTp3_distance(info->featurePose.p, info->spectrometerPose.p));
	WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(12), buf);
	if(info->pSteps>1) {
		sprintf(buf,"Cube size -- %d x %d", info->pSteps, info->tSteps);
		WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(13), buf);
	}
	else {
		sprintf(buf,"Spectral Point");
		WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(13), buf);
	}

	//WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(14), buf);
	//sprintf(buf,"Est. time: %8.2f min.",panCam_estTime);
	//WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(15), buf);
	//sprintf(buf,"Est. data vol: %f Mbits",panCam_estDataVol);
	//WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(16), buf);
}

/*****************/
```

```
/*** Spectral UI ***/
/*****************/
void BuildSpectralUI(WTui *pwtui)
{
  WTui *pwtuiSpectral;
  WTui *pwtuiMenu;
  WTui *pwtuiNirUI, *pwtuiMirUI;
  WTui *pwtuiSpectralCube;
  WTui *pwtuiParams, *pwtuiFeature;
  SpectralInfo *info;

  info = (SpectralInfo *) WTnode_getdata(spectral_node);

  pwtuiSpectral = WTuiform_new(pwtui, "Spectral",
                     WTUIATT_LEFT, (int) (ui_scale * 311.0f), WTUIATT_TOP, 0,
                     WTUIATT_WIDTH, (int) (ui_scale * 309.0f), WTUIATT_HEIGHT, (int)
(ui_scale * 275.0f), NULL);

  pwtuiMenu = WTuimenubar_new(pwtuiSpectral);

  pwtuiIdPop = WTuimenupopup_new(pwtuiMenu, "Type");
  pwtuiNirUI = WTuimenuitem_new(pwtuiIdPop, "Near IR");
  WTui_setcallback(pwtuiNirUI, WTUIEVENT_ACTIVATE, SpectralNearIR, NULL);
  pwtuiMirUI = WTuimenuitem_new(pwtuiIdPop, "Mid IR");
  WTui_setcallback(pwtuiMirUI, WTUIEVENT_ACTIVATE, SpectralMidIR, NULL);

  pwtuiExpPop = WTuimenupopup_new(pwtuiMenu, "Cube");
  pwtuiSpectralCube = WTuimenuitem_new(pwtuiExpPop, "1 x 1");
  WTui_setcallback(pwtuiSpectralCube, WTUIEVENT_ACTIVATE, SpectralCube, NULL);
  pwtuiSpectralCube = WTuimenuitem_new(pwtuiExpPop, "2 x 2");
  WTui_setcallback(pwtuiSpectralCube, WTUIEVENT_ACTIVATE, SpectralCube, NULL);
  pwtuiSpectralCube = WTuimenuitem_new(pwtuiExpPop, "3 x 3");
  WTui_setcallback(pwtuiSpectralCube, WTUIEVENT_ACTIVATE, SpectralCube, NULL);
  pwtuiSpectralCube = WTuimenuitem_new(pwtuiExpPop, "4 x 4");
  WTui_setcallback(pwtuiSpectralCube, WTUIEVENT_ACTIVATE, SpectralCube, NULL);
  pwtuiSpectralCube = WTuimenuitem_new(pwtuiExpPop, "5 x 5");
  WTui_setcallback(pwtuiSpectralCube, WTUIEVENT_ACTIVATE, SpectralCube, NULL);

  pwtuiSelectPop = WTuimenupopup_new(pwtuiMenu, "Set");
  pwtuiFeature = WTuimenuitem_new(pwtuiSelectPop, "Feature");
  pwtuiParams = WTuimenuitem_new(pwtuiSelectPop, "Params");

  WTui_setcallback(pwtuiNirUI, WTUIEVENT_ACTIVATE, SpectralNearIR, NULL);
  WTui_setcallback(pwtuiFeature, WTUIEVENT_ACTIVATE, SpectralFeature, NULL);
```

```
 WTui_setcallback(pwtuiParams, WTUIEVENT_ACTIVATE, SpectralParams, NULL);


 /*** create rendering window to display selected 2d texture image ***/
 WTmessage("Create texture form and window.\n");
 texture_form = WTui_newform(pwtuiSpectral, "2D Texture Image",
                 WTUIATT_LEFT, (int)(5.0f*ui_scale), WTUIATT_TOP, (int)((40.0f)*ui_scale),
                 WTUIATT_WIDTH, (int)(300*ui_scale),  WTUIATT_HEIGHT,
                 (int)(235.0f*ui_scale), NULL);
 texture_window = WTuiwtkwindow_new(texture_form, WTWINDOW_DEFAULT);
 WTwindow_setbgrgb(texture_window,0,0,0);
 if(texture_root!=NULL)
        WTnode_delete(texture_root);
 texture_root = WTrootnode_new();
 WTnode_setname(texture_root, "Texture Root");
 WTwindow_setrootnode(texture_window, texture_root);
 WTwindow_setfgactions(texture_window, spectral_featureTexture_display);

 WTui_manage(texture_form);
 WTui_manage(pwtuiSpectral);
}

static void SpectralNearIR(WTui *pStruct, void *pData)
{
        SpectralInfo *info;
        WTmessage("Choose Near IR.\n");
        info = (SpectralInfo *) WTnode_getdata(spectral_node);
        info->spectrometerID=NEAR_IR;
}

static void SpectralMidIR(WTui *pStruct, void *pData)
{
        SpectralInfo *info;
        WTmessage("Choose Mid IR.\n");
        info = (SpectralInfo *) WTnode_getdata(spectral_node);
        info->spectrometerID=MID_IR;
}

static void SpectralCube(WTui *pStruct, void *pData)
{
        SpectralInfo *info;
        char *cube_text;

        cube_text = WTui_gettext(pStruct);
```

```
        info = (SpectralInfo *) WTnode_getdata(spectral_node);
        if(!strcmp(cube_text,"1 x 1"))              {info->pSteps = info->tSteps = 1;}
        else if(!strcmp(cube_text,"2 x 2"))     {info->pSteps = info->tSteps = 2;}
        else if(!strcmp(cube_text,"3 x 3"))     {info->pSteps = info->tSteps = 3;}
        else if(!strcmp(cube_text,"4 x 4"))     {info->pSteps = info->tSteps = 4;}
        else if(!strcmp(cube_text,"5 x 5"))     {info->pSteps = info->tSteps = 5;}
        WTmessage("Cube size = %s\n",cube_text);
}

static void SpectralFeature(WTui *pStruct, void *pData)
{
        cursor3d.type = XHAIR;
        cursor3d.mode = INTERSECT_OBJ;
        spectral_state = SPECTRAL_SELECT_FEATURE;
        tool_activate(roverPlan_control, TRUE);
}

/*****************************/
/*** Spectral Parameter UI **/
/*****************************/
static void SpectralParams(WTui *pStruct, void *pData)
{
  WTui *pwtuiSetParams;
  WTui *pwtuiResRadio, *pwtuiResCheck;
  WTui *pwtuiClose;

  /*** Only Include The Following GUI Items on NIR ***/
  WTui *pwtuiCarbLabel;
  WTui *pwtuiAlwaysReturnCheck;
  WTui *pwtuiTestBitCheck;

  char *pacCOptions[4] = {"Mono-HiRes (Left)", "Stereo-HiRes", "Mono-LoRes (left)",
        "Stereo-LoRes"};

  WTui *pwtuiBYLabel, *pwtuiBNLabel, *pwtuiTNLabel;
  WTui *pwtuiBY1, *pwtuiBY2, *pwtuiBY3;
  WTui *pwtuiBN1, *pwtuiBN2, *pwtuiBN3;
  WTui *pwtuiTN1, *pwtuiTN2, *pwtuiTN3, *pwtuiTN4;

  SpectralInfo *info;
  int iGUIHeight = 200;

  info = (SpectralInfo *) WTnode_getdata(spectral_node);
```

```
/*** Can put in ENABLE/DISABLE on Default Spectral Box if wanted ***/
WTui_dimitem(pwtuiIdPop, TRUE);
WTui_dimitem(pwtuiExpPop, TRUE);
WTui_dimitem(pwtuiSelectPop, TRUE);

if (info->spectrometerID == NEAR_IR) {
  iGUIHeight = 650;
}

pwtuiShell = WTuiform_new(toplevel, "Set Parameters",
          WTUIATT_LEFT, (int) (ui_scale * 0.0f), WTUIATT_TOP, (int) (ui_scale * 40),
          WTUIATT_WIDTH, (int) (ui_scale * 500.0f), WTUIATT_HEIGHT, (int)
          (ui_scale * iGUIHeight), NULL);

pwtuiSetParams = WTuiform_new(pwtuiShell, "Set Parameters",
          WTUIATT_LEFT, (int) (ui_scale * 0.0f), WTUIATT_TOP, (int) (ui_scale * 0),
          WTUIATT_WIDTH, (int) (ui_scale * 500.0f), WTUIATT_HEIGHT, (int)
          (ui_scale * iGUIHeight), NULL);

pwtuiResCheck = WTuicheckbutton_new(pwtuiSetParams, "Capture Image(s) with spectral ...",
          WTUIATT_LEFT, (int) (ui_scale * 10.0f), WTUIATT_TOP, (int) (ui_scale * 10),
          WTUIATT_WIDTH, (int) (ui_scale * 525.0f), WTUIATT_HEIGHT, (int)
          (ui_scale * 15.0f), NULL);

pwtuiResRadio = WTuiradiobox_new(pwtuiSetParams, 4, pacCOptions,
          WTUIATT_LEFT, (int) (ui_scale * 25.0f), WTUIATT_TOP, (int) (ui_scale *
          35.0f), WTUIATT_WIDTH, (int) (ui_scale * 400.0f), WTUIATT_HEIGHT, (int)
          (ui_scale * 80.0f), NULL);

//  WTui_check(pwtuiResCheck, TRUE);
WTui_setselected(pwtuiResRadio, info->camID);

if (info->spectrometerID == NEAR_IR) {

  pwtuiAlwaysReturnCheck = WTuicheckbutton_new(pwtuiSetParams,
      "Return Spectral Data Always",
      WTUIATT_LEFT, (int) (ui_scale * 10.0f), WTUIATT_TOP, (int) (ui_scale * 145.0f),
      WTUIATT_WIDTH, (int) (ui_scale * 525.0f), WTUIATT_HEIGHT, (int) (ui_scale *
      20.0f), NULL);

  pwtuiTestBitCheck = WTuicheckbutton_new(pwtuiSetParams, "Test Carbonate Bit",
      WTUIATT_LEFT, (int) (ui_scale * 10.0f), WTUIATT_TOP, (int) (ui_scale * 180.0f),
      WTUIATT_WIDTH, (int) (ui_scale * 525.0f), WTUIATT_HEIGHT, (int) (ui_scale *
      20.0f), NULL);
```

```
//    WTui_check(pwtuiTestBitCheck, TRUE);

    pwtuiCarbLabel = WTuilabel_new(pwtuiSetParams, "Carbonate Bit Options", WTUI_TEXT,
          WTUIATT_LEFT, (int) (ui_scale * 10.0f), WTUIATT_TOP, (int) (ui_scale * 215.0f),
          WTUIATT_WIDTH, (int) (ui_scale * 270.0f), WTUIATT_HEIGHT, (int) (ui_scale *
          20.0f), NULL);

    pwtuiBYLabel = WTuilabel_new(pwtuiSetParams, "if Carbonate ...", WTUI_TEXT,
          WTUIATT_LEFT, (int) (ui_scale * 25.0f), WTUIATT_TOP, (int) (ui_scale * 240.0f),
          WTUIATT_WIDTH, (int) (ui_scale * 210.0f), WTUIATT_HEIGHT, (int) (ui_scale *
          20.0f), NULL);

    pwtuiBY1 = WTuicheckbutton_new(pwtuiSetParams, "Return Full Spectra",
          WTUIATT_LEFT, (int) (ui_scale * 40.0f), WTUIATT_TOP, (int) (ui_scale * 265.0f),
          WTUIATT_WIDTH, (int) (ui_scale * 255.0f), WTUIATT_HEIGHT, (int) (ui_scale *
          20.0f), NULL);

    pwtuiBY2 = WTuicheckbutton_new(pwtuiSetParams, "Capture Images",
          WTUIATT_LEFT, (int) (ui_scale * 40.0f), WTUIATT_TOP, (int) (ui_scale * 295.0f),
          WTUIATT_WIDTH, (int) (ui_scale * 255.0f), WTUIATT_HEIGHT, (int) (ui_scale *
          20.0f), NULL);

/*    pwtuiBY3 = WTuicheckbutton_new(pwtuiSetParams,
          "Traverse to Target for Close-Up Image",
          WTUIATT_LEFT, (int) (ui_scale * 40.0f), WTUIATT_TOP,  (int) (ui_scale * 325.0f),
          WTUIATT_WIDTH, (int) (ui_scale * 255.0f), WTUIATT_HEIGHT, (int) (ui_scale *
          20.0f), NULL);
*/
//    WTui_check(pwtuiBY1, TRUE);

    pwtuiBNLabel = WTuilabel_new(pwtuiSetParams, "if not Carbonate ...", WTUI_TEXT,
          WTUIATT_LEFT, (int) (ui_scale * 25.0f), WTUIATT_TOP, (int) (ui_scale * 355.0f),
          WTUIATT_WIDTH, (int) (ui_scale * 255.0f), WTUIATT_HEIGHT, (int) (ui_scale *
          20.0f), NULL);

    pwtuiBN1 = WTuicheckbutton_new(pwtuiSetParams, "Return Full Spectra",
          WTUIATT_LEFT, (int) (ui_scale * 40.0f), WTUIATT_TOP, (int) (ui_scale * 380.0f),
          WTUIATT_WIDTH, (int) (ui_scale * 255.0f), WTUIATT_HEIGHT, (int) (ui_scale *
          20.0f), NULL);

    pwtuiBN2 = WTuicheckbutton_new(pwtuiSetParams, "Capture Images",
          WTUIATT_LEFT, (int) (ui_scale * 40.0f), WTUIATT_TOP, (int) (ui_scale * 410.0f),
          WTUIATT_WIDTH, (int) (ui_scale * 255.0f), WTUIATT_HEIGHT, (int) (ui_scale *
          20.0f), NULL);
```

```
/*  pwtuiBN3 = WTuicheckbutton_new(pwtuiSetParams,
        "Traverse to Target for Close-Up Image",
        WTUIATT_LEFT, (int) (ui_scale * 40.0f), WTUIATT_TOP, (int) (ui_scale * 440.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 255.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);
                                        */
//  WTui_check(pwtuiBN1, TRUE);

    pwtuiTNLabel = WTuilabel_new(pwtuiSetParams, "if too noisy ...", WTUI_TEXT,
        WTUIATT_LEFT, (int) (ui_scale * 25.0f), WTUIATT_TOP, (int) (ui_scale * 470.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 210.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);

    pwtuiTN1 = WTuicheckbutton_new(pwtuiSetParams, "Re-Acquire",
        WTUIATT_LEFT, (int) (ui_scale * 40.0f), WTUIATT_TOP, (int) (ui_scale * 495.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 150.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);

    pwtuiTN2 = WTuicheckbutton_new(pwtuiSetParams, "Return Noisy Full Spectra",
        WTUIATT_LEFT, (int) (ui_scale * 40.0f), WTUIATT_TOP, (int) (ui_scale * 525.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 300.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);

    pwtuiTN3 = WTuicheckbutton_new(pwtuiSetParams, "Capture Noisy Images",
        WTUIATT_LEFT, (int) (ui_scale * 40.0f), WTUIATT_TOP, (int) (ui_scale * 555.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 225.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);

/*      pwtuiTN4 = WTuicheckbutton_new(pwtuiSetParams,
        "Traverse to Target for Close-Up Image",
        WTUIATT_LEFT, (int) (ui_scale * 40.0f), WTUIATT_TOP, (int) (ui_scale * 585.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 555.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);
*/
//  WTui_check(pwtuiTN1, TRUE);
    WTui_setcallback(pwtuiAlwaysReturnCheck, WTUIEVENT_ACTIVATE,
        ParamsAlwaysReturn, NULL);
    WTui_setcallback(pwtuiTestBitCheck, WTUIEVENT_ACTIVATE, ParamsTestBit, NULL);
    WTui_setcallback(pwtuiBY1, WTUIEVENT_ACTIVATE, ParamsICCompare, NULL);
    WTui_setcallback(pwtuiBY2, WTUIEVENT_ACTIVATE, ParamsICCompare, NULL);
//  WTui_setcallback(pwtuiBY3, WTUIEVENT_ACTIVATE, ParamsICCompare, NULL);
    WTui_setcallback(pwtuiBN1, WTUIEVENT_ACTIVATE, ParamsNCCompare, NULL);
    WTui_setcallback(pwtuiBN2, WTUIEVENT_ACTIVATE, ParamsNCCompare, NULL);
//  WTui_setcallback(pwtuiBN3, WTUIEVENT_ACTIVATE, ParamsNCCompare, NULL);
```

```
    WTui_setcallback(pwtuiTN1, WTUIEVENT_ACTIVATE, ParamsTNCompare, NULL);
    WTui_setcallback(pwtuiTN2, WTUIEVENT_ACTIVATE, ParamsTNCompare, NULL);
    WTui_setcallback(pwtuiTN3, WTUIEVENT_ACTIVATE, ParamsTNCompare, NULL);
//    WTui_setcallback(pwtuiTN4, WTUIEVENT_ACTIVATE, ParamsTNCompare, NULL);
  }

  pwtuiClose = WTuipushbutton_new(pwtuiSetParams, "Close",
          WTUIATT_LEFT, (int) (ui_scale * 10.0f), WTUIATT_TOP, (int) (ui_scale *
          (iGUIHeight - 45.0f)), WTUIATT_WIDTH, (int) (ui_scale * 150.0f),
          WTUIATT_HEIGHT, (int) (ui_scale * 40.0f), NULL);

  WTui_setcallback(pwtuiResCheck, WTUIEVENT_ACTIVATE, ParamsDoCapture, NULL);
  WTui_setcallback(pwtuiResRadio, WTUIEVENT_ACTIVATE, ParamsCameraSelect, NULL);

  WTui_setcallback(pwtuiClose, WTUIEVENT_ACTIVATE, SetParamsClose, NULL);

  WTui_manage(pwtuiSetParams);
  WTui_manage(pwtuiShell);
}

static void ParamsAlwaysReturn(WTui *pStruct, void *pData)
{
  SpectralInfo *info;

  WTmessage("Toggle Return Full Spectra Always Option.\n");
  info = (SpectralInfo *) WTnode_getdata(spectral_node);

  info->returnSpectra = WTui_checkbuttonstate(pStruct);
}

static void ParamsTestBit(WTui *pStruct, void *pData)
{
  SpectralInfo *info;

  WTmessage("Toggle Carbonate Test Option.\n");
  info = (SpectralInfo *) WTnode_getdata(spectral_node);

  info->testCarbonate = WTui_checkbuttonstate(pStruct);
}

static void ParamsICCompare(WTui *pStruct, void *pData)
{
  SpectralInfo *info;
  char *text;
```

```
  text = WTui_gettext(pStruct);

  info = (SpectralInfo *) WTnode_getdata(spectral_node);

  if (!strcmp(text, "Capture Images")) {
          info->ic_hiResImageFlag = Wtui_checkbuttonstate(pStruct);
  } else if (!strcmp(text, "Return Full Spectra")) {
          info->ic_spectraDataFlag = Wtui_checkbuttonstate(pStruct);
  } else if (!strcmp(text, "Traverse to Target for Close-Up Image")) {
          info->ic_traverseFlag = Wtui_checkbuttonstate(pStruct);
  }
}

static void ParamsNCCompare(WTui *pStruct, void *pData)
{
  SpectralInfo *info;
  char *text;

  text = WTui_gettext(pStruct);

  info = (SpectralInfo *) WTnode_getdata(spectral_node);

  if (!strcmp(text, "Capture Images")) {
          info->nc_hiResImageFlag = Wtui_checkbuttonstate(pStruct);
  } else if (!strcmp(text, "Return Full Spectra")) {
          info->nc_spectraDataFlag = Wtui_checkbuttonstate(pStruct);
  } else if (!strcmp(text, "Traverse to Target for Close-Up Image")) {
          info->nc_traverseFlag = Wtui_checkbuttonstate(pStruct);
  }
}

static void ParamsTNCompare(WTui *pStruct, void *pData)
{
  SpectralInfo *info;
  char *text;

  text = WTui_gettext(pStruct);

  info = (SpectralInfo *) WTnode_getdata(spectral_node);

  if (!strcmp(text, "Capture Noisy Images")) {
          info->tn_hiResImageFlag = Wtui_checkbuttonstate(pStruct);
  }else if (!strcmp(text, "Return Noisy Full Spectra")) {
          info->tn_spectraDataFlag = Wtui_checkbuttonstate(pStruct);
```

```
  } else if (!strcmp(text, "Re-Acquire")) {
          info->tn_reAcquireFlag = Wtui_checkbuttonstate(pStruct);
  }else if (!strcmp(text, "Traverse to Target for Close-Up Image")) {
          info->tn_traverseFlag = Wtui_checkbuttonstate(pStruct);
  }
}

static void ParamsDoCapture(WTui *pStruct, void *pData)
{
          SpectralInfo *info;

          WTmessage("Toggle Capture Image(s).\n");
          info = (SpectralInfo *) WTnode_getdata(spectral_node);

          /*** If a camera was selected, it won't image, otherwise, it is set to the camera ID ***/
          if (WTui_checkbuttonstate(pStruct) == 0)
            info->camID = 0;
          else info->camID = WTui_getselected(pStruct);
}

static void ParamsCameraSelect(WTui *pStruct, void *pData)
{
          SpectralInfo *info;

          WTmessage("Camera Select.\n");
          info = (SpectralInfo *) WTnode_getdata(spectral_node);

          info->camID = WTui_getselected(pStruct);
}

static void SetParamsClose(WTui *pStruct, void *pData)
{
  WTui_dimitem(pwtuiIdPop, FALSE);
  WTui_dimitem(pwtuiExpPop, FALSE);
  WTui_dimitem(pwtuiSelectPop, FALSE);

  WTui_delete(pwtuiShell);
}
```

Appendix D

Navigation Camera Planner Source Code

This is a complete listing of the source code (in C) for the navigation camera planner. After the primary implementation by Allport, additions were made by Ted Blackmon. Blackmon's changes are included.

D.1 Listing of navCam.h

```
/***************************************************************************
**
** navCam.h - planning module for navigation Image camera for Marsokhod rover
**
****************************************************************************/
void navCam_init();
void navCam_loop();
void navCam_exit();
WTnode *navCam_newTask(WTnode *);
void navCam_editTask(WTnode *);
void navCam_seq(WTnode *, FILE *file);
```

D.2 Listing of navCam.c

```
/***************************************************************************
** navCam.c - planning module for Navigation Image camera for Marsokhod rover
**
** Written by Chris Allport & Theodore T. Blackmon
** Copyright 1999.
```

```
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <assert.h>

#include "wt.h"
#include "ims.h"
#include "overlay.h"
#include "text3d.h"
#include "view.h"

#include "rover_plan.h"
#include "marsokhod.h"

/**********************************************************/

void navCam_init();
void navCam_loop();
void navCam_exit();
WTnode *navCam_newTask(WTnode *);
void navCam_editTask(WTnode *);
void navCam_seq(WTnode *, FILE *);

/**********************************************************/

typedef struct {
  //sGenericTaskInfo gen;

  int camID;
  int resolution;
  int compression;
  WTpq camera_pose_fl;
  WTpq camera_pose_fr;
  WTpq camera_pose_rl;
  WTpq camera_pose_rr;

} NavCamInfo;


#define CAM_X_FOV                20
#define CAM_Y_FOV                15
```

```
#define FRONT_LEFT_CAM                26
#define FRONT_RIGHT_CAM               27
#define FRONT_STEREO_CAM      29
#define REAR_LEFT_CAM         30
#define REAR_RIGHT_CAM                31
#define REAR_STEREO_CAM               33

/*** Default Definitions ***/
#define NAV_CAM_DEFAULT               FRONT_LEFT_CAM
#define RESOLUTION_DEFAULT      1
#define COMPRESSION_DEFAULT     8

static char navCamId_text[256];

/*** nav-cam nodes ************************************************/
static WTnode *navCam_taskNode;
static WTnode *navCam_node;


/*** Nav Cam UI ***************************************************/
void BuildNavCamUI(WTui *pwtui);
static void NavCamCamSelect(WTui *pStruct, void *pData);
static void NavCamResolution(WTui *pStruct, void *pData);
static void NavCamCompression(WTui *pStruct, void *pData);
static void NavCamSnapCam(WTui *pStruct, void *pData);


/*** 2d & 3D Overays Variables ************************************/
static float fNavCamProjectionDistance = 2.0f;
void make_3d_projection(WTwindow *w, int camID);
static void navCam_image_projections(WTwindow *w, FLAG eye);
static void navCam_plan_readout(WTwindow *w, FLAG eye);

/*** Variables for time and data volume stats ***/
static float navCam_estDataVol;             /*** in mBits ***/
#define NAVCAM_TIME_PER_PT                  0.1f
#define NAVCAM_MBITS_PER_IMAGE              2.4

/*****************************************************************/
/** Initialization function for nav cam planning              ***/
/*****************************************************************/
void navCam_init()
{
        WTui *ui;
```

```
        WTmessage("Nav Cam Init Function Called\n");
        ui = BuildGeneralUI();
        BuildNavCamUI(ui);
        WTui_manage(ui);

        /*** add 3d overlay for navigation image projection window ***/
        overlay3d_add(navCam_image_projections);
        overlay2d_add(navCam_plan_readout);
}

void navCam_loop()
{
}

void navCam_exit()
{
        /*** remove 3d/2d overlay for navigation image projection window ***/
        overlay3d_delete(navCam_image_projections);
        overlay2d_delete(navCam_plan_readout);
        WTmessage("Leaving Nav Can\n");
}

WTnode *navCam_newTask(WTnode *marso)
{
        WTnode *xform;
        NavCamInfo *info;
        WTp3 pos;
        WTq rot, q1,q2,q3;
        WTnodepath *pwtnpRover;
        marsokhod_model_info *MarsoInfo;

        WTmessage("Create a new nav-cam node!\n");

        /*** make a sep, xform, and geom node ***/
        navCam_taskNode = WTsepnode_new(NULL);
        xform = WTxformnode_new(navCam_taskNode);
        pos[X] = 0.0f;
        pos[Y] = 0.0f;
        pos[Z] = -1.6f;
        WTnode_settranslation(xform, pos);
        navCam_node = WTgeometrynode_new(navCam_taskNode,
                        WTgeometry_newcylinder(2.0, 0.1, 8, FALSE, TRUE));
        WTnode_enable(navCam_node, FALSE);
```

```
/*** allocate memory for nav-cam info struct ***/
info = (NavCamInfo *)malloc(sizeof(NavCamInfo));
if (info == NULL) printf("malloc failed for NavCamInfo struct.\n");

info->camID = NAV_CAM_DEFAULT;
info->resolution = RESOLUTION_DEFAULT;
info->compression = COMPRESSION_DEFAULT;
sprintf(navCamId_text,"Front Left");

/*** Set Rover Camera Position ***/
MarsoInfo = (marsokhod_model_info *) WTnode_getdata(WTnode_getchild(marso,0));
pwtnpRover = WTnodepath_new(
        WTnode_getchild(MarsoInfo->frontaxle, 0), ims_root, 0);
WTnodepath_gettranslation(pwtnpRover, pos);
WTnodepath_getorientation(pwtnpRover, rot);

WTeuler_2q(0.0f, radians(90.0f), 0.0f, q1);
WTeuler_2q(radians(90.0f), 0.0f, 0.0f, q2);
WTq_mult(q1, q2, q3);
WTeuler_2q(0.0f, 0.0f, radians(180.0f), q1);
WTq_mult(q1, q3, q3);
WTq_mult(q3, rot, rot);

WTnodepath_delete(pwtnpRover);
WTp3_copy(pos, info->camera_pose_fl.p);
WTq_copy(rot, info->camera_pose_fl.q);
WTp3_copy(pos, info->camera_pose_fr.p);
WTq_copy(rot, info->camera_pose_fr.q);

MarsoInfo = (marsokhod_model_info *) WTnode_getdata(WTnode_getchild(marso,0));
pwtnpRover = Wtnodepath_new(
        WTnode_getchild(MarsoInfo->rearaxle, 0), ims_root, 0);
WTnodepath_gettranslation(pwtnpRover, pos);
WTnodepath_getorientation(pwtnpRover, rot);

WTeuler_2q(0.0f, radians(90.0f), 0.0f, q1);
WTeuler_2q(radians(90.0f), 0.0f, 0.0f, q2);
WTq_mult(q1, q2, q3);
WTeuler_2q(0.0f, 0.0f, radians(180.0f), q1);
WTq_mult(q1, q3, q3);
WTq_mult(q3, rot, rot);

WTnodepath_delete(pwtnpRover);
WTp3_copy(pos, info->camera_pose_rl.p);
```

```
	WTq_copy(rot, info->camera_pose_rl.q);
	WTp3_copy(pos, info->camera_pose_rr.p);
	WTq_copy(rot, info->camera_pose_rr.q);

	pos[X] = -0.1f; pos[Y] = -0.25f; pos[Z] = 0.55f;
	WTp3_local2worldframe(pos, &info->camera_pose_fl, info->camera_pose_fl.p);

	pos[X] = 0.1f; pos[Y] = -0.25f; pos[Z] = 0.55f;
	WTp3_local2worldframe(pos, &info->camera_pose_fr, info->camera_pose_fr.p);

	pos[X] = -0.1f; pos[Y] = -0.25f; pos[Z] = -0.55f;
	WTp3_local2worldframe(pos, &info->camera_pose_rl, info->camera_pose_rl.p);

	pos[X] = 0.1f; pos[Y] = -0.25f; pos[Z] = -0.55f;
	WTp3_local2worldframe(pos, &info->camera_pose_rr, info->camera_pose_rr.p);

	/*** set default value of nav-cam info struct ***/
	WTnode_setdata(navCam_node, (void *)info);

	return(navCam_taskNode);
}

void navCam_editTask(WTnode *node)
{
	WTmessage("Edit nav-cam node!\n");
	navCam_node = WTnode_getchild(node, 1);
}

/*****************************************************************
***  Output navCam arameter list to crl-g file
*****************************************************************/
void navCam_seq(WTnode *node, FILE *file)
{
	NavCamInfo *info;
	navCam_node = WTnode_getchild(node, 1);
	info = (NavCamInfo *) WTnode_getdata(navCam_node);

	fprintf(file, "	;for navCam\n");
	fprintf(file, "	:camId %d\n",info->camID);
	fprintf(file, "	:resolution %d\n",info->resolution);
	fprintf(file, "	:compression %d\n",info->compression);
}

/*****************************************************************/
```

```
/*** 3D overlay function to draw navigation image projections                    ***/
/*****************************************************************/
static void build_3d_projections(WTwindow *w, int camID)
{
  int i;
  float x, y;
  NavCamInfo *info;
  WTp3 vect[8];
  WTp3 p;
  WTpq pose;
  int iDir;

  x = radians(CAM_X_FOV / 2);
  y = radians(CAM_Y_FOV / 2);

  info = (NavCamInfo *) WTnode_getdata(navCam_node);

  switch (camID) {
    case FRONT_LEFT_CAM :
      WTwindow_set3Dcolor(w, 255, 0, 0);
      WTpq_copy(&info->camera_pose_fl, &pose);
      iDir = 1;
      break;
    case FRONT_RIGHT_CAM :
      WTwindow_set3Dcolor(w, 0, 255, 0);
      WTpq_copy(&info->camera_pose_fr, &pose);
      iDir = 1;
      break;
    case REAR_LEFT_CAM :
      WTwindow_set3Dcolor(w, 0, 0, 255);
      WTpq_copy(&info->camera_pose_rl, &pose);
      iDir = -1;
      break;
    case REAR_RIGHT_CAM :
      WTwindow_set3Dcolor(w, 0, 0, 0);
      WTpq_copy(&info->camera_pose_rr, &pose);
      iDir = -1;
      break;
    default :
      return;
  }

  for (i = 0; i < 4; i ++) {
    WTp3_copy(pose.p, vect[2 * i]);
```

```
 }

 /*** Front Left ***/
 p[X] = iDir * fNavCamProjectionDistance * sin(-x) * cos(-y);
 p[Y] = iDir * fNavCamProjectionDistance * sin(-y);
 p[Z] = iDir * fNavCamProjectionDistance * cos(-x) * cos(-y);
 WTp3_local2worldframe(p, &pose, p);
 WTp3_copy(p, vect[1]);

 /*** Front Right ***/
 p[X] = iDir * fNavCamProjectionDistance * sin(x) * cos(-y);
 p[Y] = iDir * fNavCamProjectionDistance * sin(-y);
 p[Z] = iDir * fNavCamProjectionDistance * cos(x) * cos(-y);
 WTp3_local2worldframe(p, &pose, p);
 WTp3_copy(p, vect[3]);

 /*** Rear Right ***/
 p[X] = iDir * fNavCamProjectionDistance * sin(x) * cos(y);
 p[Y] = iDir * fNavCamProjectionDistance * sin(y);
 p[Z] = iDir * fNavCamProjectionDistance * cos(x) * cos(y);
 WTp3_local2worldframe(p, &pose, p);
 WTp3_copy(p, vect[5]);

 /*** Rear Left ***/
 p[X] = iDir * fNavCamProjectionDistance * sin(-x) * cos(y);
 p[Y] = iDir * fNavCamProjectionDistance * sin(y);
 p[Z] = iDir * fNavCamProjectionDistance * cos(-x) * cos(y);
 WTp3_local2worldframe(p, &pose, p);
 WTp3_copy(p, vect[7]);

 WTwindow_draw3Dlines(w, vect, 8, WTLINE_SEGMENTS);

 WTp3_copy(vect[1], vect[0]);
 WTp3_copy(vect[3], vect[1]);
 WTp3_copy(vect[5], vect[2]);
 WTp3_copy(vect[7], vect[3]);
 WTwindow_draw3Dlines(w, vect, 4, WTLINE_CLOSE);
}

static void navCam_image_projections(WTwindow *w, FLAG eye)
{
 NavCamInfo *info;

 info = (NavCamInfo *) WTnode_getdata(navCam_node);
```

```
  switch (info->camID) {
    case FRONT_LEFT_CAM :
      build_3d_projections(w, FRONT_LEFT_CAM);
      break;
    case FRONT_RIGHT_CAM :
      build_3d_projections(w, FRONT_RIGHT_CAM);
      break;
    case FRONT_STEREO_CAM :
      build_3d_projections(w, FRONT_LEFT_CAM);
      build_3d_projections(w, FRONT_RIGHT_CAM);
      break;
    case REAR_LEFT_CAM :
      build_3d_projections(w, REAR_LEFT_CAM);
      break;
    case REAR_RIGHT_CAM :
      build_3d_projections(w, REAR_RIGHT_CAM);
      break;
    case REAR_STEREO_CAM :
      build_3d_projections(w, REAR_LEFT_CAM);
      build_3d_projections(w, REAR_RIGHT_CAM);
      break;
    default :
      return;
  }

}

/*******************************************************************
** 2D overlay function to readout panorama parameters
*******************************************************************/

static void navCam_plan_readout(WTwindow *w, FLAG eye)
{
	int i=1;
	char buf[256];
	NavCamInfo *info;

	info = (NavCamInfo *) WTnode_getdata(navCam_node);
	if(info==NULL) return;
	WTwindow_set2Dfont(w,1);
	WTwindow_set2Dcolor(window_id, font_color[0], font_color[1], font_color[2]);
	sprintf(buf,"camera ID -- %d (%s)",info->camID,navCamId_text);
	WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(7), buf);
	sprintf(buf,"resolution 1/%d",info->resolution);
```

```
        WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(8), buf);
        sprintf(buf,"compression %d:1",info->compression);
        WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(9), buf);

        /*** Compute data volume and time statistics ***/
        navCam_estDataVol = NAVCAM_MBITS_PER_IMAGE;
        if(info->camID == FRONT_STEREO_CAM || info->camID == REAR_STEREO_CAM)
                navCam_estDataVol += navCam_estDataVol;

        /*** factor in resolution ***/
        navCam_estDataVol = navCam_estDataVol/(info->resolution*info->resolution);
        /*** factor in compression ***/
        navCam_estDataVol = navCam_estDataVol/(info->compression);

        sprintf(buf,"Est. time: %8.2f min.",NAVCAM_TIME_PER_PT);
        WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(10), buf);
        sprintf(buf,"Est. data vol: %f Mbits",navCam_estDataVol);
        WTwindow_draw2Dtext(window_id, 0.025f, 0.95f- 0.025*(float)(11), buf);
}

/******************************************************************************
*** Nav Cam UI
******************************************************************************/
void BuildNavCamUI(WTui *pwtui)
{
  WTui *pwtuiNavCam;
  WTui *pwtuiCamIDLabel, *pwtuiCamRadio;
  WTui *pwtuiMenu;
  WTui *pwtuiResPop, *pwtuiComPop;
  WTui *Res, *Comp;
  WTui *pwtuiSnapCamButton;
  char *pacNavCamRadio[6] = {       "Front Left",
                                    "Front Right",
                                    "Front Stereo",
                                    "Rear Left",
                                    "Rear Right",
                                    "Rear Stereo"};        // 6 Choices
  NavCamInfo *info;

  info = (NavCamInfo *) WTnode_getdata(navCam_node);

  pwtuiNavCam = WTuiform_new(pwtui, "Nav Cam",
        WTUIATT_LEFT, (int) (ui_scale * 311.0f), WTUIATT_TOP, 0,
        WTUIATT_WIDTH, (int) (ui_scale * 309.0f), WTUIATT_HEIGHT, (int) (ui_scale *
```

```
        275.0f), NULL);

pwtuiCamIDLabel = WTuilabel_new(pwtuiNavCam, "Navigation Camera", WTUI_TEXT,
        WTUIATT_LEFT, (int) (ui_scale * 50.0f), WTUIATT_TOP, (int) (ui_scale * 50.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 200.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        20.0f), NULL);

pwtuiCamRadio = WTuiradiobox_new(pwtuiNavCam, 6, pacNavCamRadio,
        WTUIATT_LEFT, (int) (ui_scale * 60.0f), WTUIATT_TOP, (int) (ui_scale * 80.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 150.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        80.0f), NULL);
WTui_setcallback(pwtuiCamRadio, WTUIEVENT_ACTIVATE, NavCamCamSelect, NULL);

WTui_setselected(pwtuiCamRadio, info->camID - 26);

pwtuiMenu = WTuimenubar_new(pwtuiNavCam);
pwtuiResPop = WTuimenupopup_new(pwtuiMenu, "Res");
Res = WTuimenuitem_new(pwtuiResPop, "1/1");
WTui_setcallback(Res, WTUIEVENT_ACTIVATE, NavCamResolution, NULL);
Res = WTuimenuitem_new(pwtuiResPop, "1/2");
WTui_setcallback(Res, WTUIEVENT_ACTIVATE, NavCamResolution, NULL);
Res = WTuimenuitem_new(pwtuiResPop, "1/3");
WTui_setcallback(Res, WTUIEVENT_ACTIVATE, NavCamResolution, NULL);
Res = WTuimenuitem_new(pwtuiResPop, "1/4");
WTui_setcallback(Res, WTUIEVENT_ACTIVATE, NavCamResolution, NULL);

pwtuiComPop = WTuimenupopup_new(pwtuiMenu, "Comp");
Comp = WTuimenuitem_new(pwtuiComPop, "1:1");
WTui_setcallback(Comp, WTUIEVENT_ACTIVATE, NavCamCompression, NULL);
Comp = WTuimenuitem_new(pwtuiComPop, "8:1");
WTui_setcallback(Comp, WTUIEVENT_ACTIVATE, NavCamCompression, NULL);

Comp = WTuimenuitem_new(pwtuiComPop, "16:1");
WTui_setcallback(Comp, WTUIEVENT_ACTIVATE, NavCamCompression, NULL);
Comp = WTuimenuitem_new(pwtuiComPop, "32:1");
WTui_setcallback(Comp, WTUIEVENT_ACTIVATE, NavCamCompression, NULL);


pwtuiSnapCamButton = WTuipushbutton_new(pwtuiNavCam, "Snap View to Camera",
        WTUIATT_LEFT, (int) (ui_scale * 25.0f), WTUIATT_TOP, (int) (ui_scale * 240.0f),
        WTUIATT_WIDTH, (int) (ui_scale * 250.0f), WTUIATT_HEIGHT, (int) (ui_scale *
        25.0f), NULL);

WTui_setcallback(pwtuiCamRadio, WTUIEVENT_ACTIVATE, NavCamCamSelect, NULL);
```

```
  WTui_setcallback(pwtuiSnapCamButton, WTUIEVENT_ACTIVATE, NavCamSnapCam,
          NULL);

  WTui_manage(pwtuiNavCam);
}

static void NavCamCamSelect(WTui *pStruct, void *pData)
{
  int buttonNum;
  NavCamInfo *info;

  info = (NavCamInfo *) WTnode_getdata(navCam_node);

  buttonNum = WTui_getselected(pStruct);
  switch(buttonNum)  {
          case 0: info->camID=FRONT_LEFT_CAM;
                    sprintf(navCamId_text,"Front Left");
                    break;
          case 1: info->camID=FRONT_RIGHT_CAM;
                    sprintf(navCamId_text,"Front Right");
                    break;
          case 2: info->camID=FRONT_STEREO_CAM;
                    sprintf(navCamId_text,"Front Stereo");
                    break;
          case 3: info->camID=REAR_LEFT_CAM;
                    sprintf(navCamId_text,"Rear Left");
                    break;
          case 4: info->camID=REAR_RIGHT_CAM;
                    sprintf(navCamId_text,"Rear Right");
                    break;
          case 5: info->camID=REAR_STEREO_CAM;
                    sprintf(navCamId_text,"Rear Stereo");
                     break;
  }
}

static void NavCamResolution(WTui *pStruct, void *pData)
{
  NavCamInfo *info;
  char *res_text;

  res_text = WTui_gettext(pStruct);
  info = (NavCamInfo *) WTnode_getdata(navCam_node);
  if(!strcmp(res_text,"1/1"))              {info->resolution=1;}
```

```
  else if(!strcmp(res_text,"1/2"))        {info->resolution=2;}
  else if(!strcmp(res_text,"1/3"))        {info->resolution=3;}
  else if(!strcmp(res_text,"1/4"))        {info->resolution=4;}
  WTmessage("Resolution = %s (%d)\n",res_text, info->resolution);
}

static void NavCamCompression(WTui *pStruct, void *pData)
{
  NavCamInfo *info;
  char *comp_text;

  comp_text = WTui_gettext(pStruct);
  info = (NavCamInfo *) WTnode_getdata(navCam_node);
  if(!strcmp(comp_text,"1:1"))           {info->compression=1;}
  else if(!strcmp(comp_text,"8:1"))      {info->compression=8;}
  else if(!strcmp(comp_text,"16:1"))   {info->compression=16;}
  else if(!strcmp(comp_text,"32:1"))   {info->compression=32;}
  WTmessage("Compression = %s (%d)\n",comp_text, info->compression);
}

static void NavCamSnapCam(WTui *pStruct, void *pData)
{
  NavCamInfo *info;
  WTpq pose;
  WTq q;

  WTeuler_2q(0.0f, radians(180.0f), 0.0f, q);

  info = (NavCamInfo *) WTnode_getdata(navCam_node);

  switch (info->camID) {
  case FRONT_LEFT_CAM :
    WTq_copy(info->camera_pose_fl.q, pose.q);
    WTp3_copy(info->camera_pose_fl.p, pose.p);
    break;
  case FRONT_RIGHT_CAM :
    WTq_copy(info->camera_pose_fr.q, pose.q);
    WTp3_copy(info->camera_pose_fr.p, pose.p);
    break;
  case FRONT_STEREO_CAM :
    WTq_copy(info->camera_pose_fl.q, pose.q);
    WTp3_add(info->camera_pose_fl.p, info->camera_pose_fr.p, pose.p);
    WTp3_mults(pose.p, 0.5f);
    break;
```

```
  case REAR_LEFT_CAM :
    WTq_copy(info->camera_pose_rl.q, pose.q);
    WTp3_copy(info->camera_pose_rl.p, pose.p);
    WTq_mult(q, pose.q, pose.q);
    break;
  case REAR_RIGHT_CAM :
    WTq_copy(info->camera_pose_rr.q, pose.q);
    WTp3_copy(info->camera_pose_rr.p, pose.p);
    WTq_mult(q, pose.q, pose.q);
    break;
  case REAR_STEREO_CAM :
    WTq_copy(info->camera_pose_rl.q, pose.q);
    WTp3_add(info->camera_pose_rl.p, info->camera_pose_rr.p, pose.p);
    WTp3_mults(pose.p, 0.5f);
    WTq_mult(q, pose.q, pose.q);
    break;
  }

  view_follow_rover = 0;

  set_spacecraft_view(&pose);

}
```

Appendix E

Terrain Following Source Code

This is a complete listing of the source code (in C) for Marsokhod's terrain following algorithms.

E.1 Listing of marsokhod.h

```
/****************************************************************************
** marsokhod.h - Header file for Marsokhod rover.
**
** Written by Theodore T. Blackmon & Chris Allport
** Copyright 1998.
****************************************************************************/
extern WTui *marsoGo1;
#define MARSO_MODEL_SCALE 1.0

// Numerical indexing - makes certain assignments easier
// To access Right Front Wheel, use MARSO_RIGHT + MARSO_FRONT
#define MARSO_LEFT          0
#define MARSO_RIGHT         3
#define MARSO_FRONT         0
#define MARSO_MIDDLE        1
#define MARSO_REAR          2

// Straight Number Wheel Indexing
#define MARSO_LF        0
#define MARSO_LM        1
#define MARSO_LR        2
#define MARSO_RF        3
#define MARSO_RM        4
#define MARSO_RR        5

// Axle Designations for Easier Array Indexing
```

```
#define FRONT_AXLE          0
#define REAR_AXLE           1
#define MIDDLE_AXLE         2

// Wheel Radii
#define MARSO_WHEEL_RAD             0.16
#define MARSO_WHEEL_RADINNER        0.10

// Distances from Center of Rover to point on Wheel
#define MARSO_INNER_RAD             0.07
#define MARSO_CENTER_RAD            0.30
#define MARSO_OUTER_RAD             0.48

// Driving Constants borrowed from Sojo
#define STEERANG        45
#define ROVWIDTH        1.0

// Drive Command Defs
#define MARSO_STOP          0
#define MARSO_DRIVE         1
#define MARSO_RESET         2
#define MARSO_MOVE          3


// Marsokhod Model Information Structure
typedef struct {
        char label[256];
        WTp3 pos;                       // Position of Rover
        WTp3 rot;                       // Heading of Rover
        WTnode *marso_xform;            // Transform Node of Rover

 /*     WTnodepath *camera_path[3]; */  // Doesn't appear to be needed -- not used
        WTnode *camera_xform;           // Rover Camera Transform Node

        WTnode *axle_xform[3];          // Xform nodes of front/middle/rear axles
        WTnode *wheel_xform[6];         // Xform nodes of the Six Wheels
        WTnode *frontaxle, *middleaxle, *rearaxle;      // Sep Node of the front/middle/rear axles
        WTnode *arm_1, *arm_2, *arm_3, *arm_4, *arm_5, *arm_6;  // Xform Nodes of
                                                                // MacDac Arm Pieces
        WTnode *lf, *lm, *lr;           // Sep Nodes of the Left Wheels Front/Middle/Rear
        WTnode *rf, *rm, *rr;           // Sep Nodes of the Right Wheels Front/Middle/Rear

        WTnode *pantilt;                // Xform of Pantilt
        WTnode *xray;                   // Xform of Xray Unit
```

```
        WTnode *carousel;                   // Xform of Carousel

        float middle_height;                // Height of middle axle
        float axle_angle[3];                // Angle (Pitch) of Front/Rear Axle to Middle Axle
        float wheel_angle[3];               // Angle (Roll) of Front/Rear/Middle Axles

        float arm_angle[5];                 // Angles of Arm Pieces

        int DriveCommand;           /* tells rover what to do: stop, fwd, back & left, right */
        float DriveRate;            /* units per simulation step to move */
        float TurnRate;             /* degs per simulation step to turn */

        int CameraCommand;          /* tells rover how to move cam: stop,fwd,back,left, right */
        float CameraPanRate;        /* amt per simulation step to turn */
        float CameraTiltRate;       /* amt per simulation step to turn */

} marsokhod_model_info;


extern int view_follow_rover;
extern int roverOps;

WTnode *marsoNode_new(char roverLabel);


/*** Functions in marsokhod_cad.c ***/
void marsokhod_parts_init();
WTnode *marsokhod_model_add(WTp3 marso_pos, WTp3 marso_rot, char *marso_label);
WTnode *marsokhod_arm_model_add(WTp3 marso_pos, WTp3 marso_rot, char *marso_label);
void marsokhod_panCam_xform(WTnode *marso, WTpq *pose);
void home_arm(WTnode *marso_node);

/*** Callback Function for Marsokhod Drive Panel ***/
void marso_panel_fnc(WTui *ui, void *pData);

/*** marsokhod_control functions ***/
void marsokhod_init();
void marsokhod_control();
void marsokhod_exit();

/*** Terrain following rover ***/
void marsokhod_terrainSim(WTnode *marso_node);
```

E.2 Listing of marsokhod_sim.h

```
/*************************************************************************
** marsokhod_sim.h - Marsmap Model for Simulation of Marsokhod rover
**
** Written by Christopher S. Allport & Ted Blackmon
** Copyright 1998.
*************************************************************************/
#define MIDDLE_HEIGHT   0.5
#define TOLERANCE       0.001;

char marsokhod_model_elevation(WTnode *marso_node, float height[6]);
void settle_marso(WTnode *marso_node);
```

E.3 Listing of marsokhod_sim.c

```
/*************************************************************************
**
** marsokhod_sim.c - Marsmap Model for Simulation of Marsokhod rover
**
** Written by Christopher S. Allport & Ted Blackmon
** Copyright 1998.
*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#include "wt.h"
#include "ims.h"
#include "marsokhod.h"
#include "marsokhod_sim.h"

/*****************************************************************
*** Public variables and function prototypes
*****************************************************************/
extern WTnode *marsmap;
extern marsokhod_model_info *marso_info;
extern FLAG fFunky;

char marsokhod_model_elevation(WTnode *marso_node, float height[6]);
```

```
/*******************************************************************
*** Private variables and function prototypes
*******************************************************************/
#define MARSO_WHEEL_BASE     0.9
#define FB_OFFSET            0.5
#define WHEEL_OFFSET         0.6

#define EDGE_TSCALAR         0.22
#define AXLE_TSCALAR         0.22

#define EDGE_NORMAL          2
#define AXLE_NORMAL          1
#define NO_NORMAL            0

void settle_marso(WTnode *marso_node);
void marsokhod_model_update(WTnode *marso_node);
void marsokhod_arm_update(WTnode *marso_node);

void marsokhod_model_simStep(WTnode *marso_node)
{
}

void marsokhod_terrainSim(WTnode *marso_node)
{
	settle_marso(marso_node);
}

/************************************************************************
*** Update marsokhod node state
************************************************************************/
void marsokhod_model_update(WTnode *marso_node)
{
	marsokhod_model_info *model_info;
	WTpq pose;
	WTq q, qq;
	WTq qx,qy,qz;
	WTp3 pos;

	/*** get internal data structure for marsokhod rover ***/
	model_info = (marsokhod_model_info *) WTnode_getdata(marso_node);

	/*** position and rotation of vehicle chasis ***/
	/*** rotate marso node to align with ims coordinate system ***/
	WTq_init(q);
```

```
WTnode_setorientation(model_info->marso_xform, q);
WTnode_rotate(model_info->marso_xform, 0.0f, -90.0f, 180.0f, WTFRAME_PARENT);

/*** set marsokhod position***/
WTp3_init(pos);
WTnode_settranslation(model_info->marso_xform, pos);

/*** set marsokhod position in Mars Coordinates***/
pos[X] = model_info->pos[Y];
pos[Y] = model_info->pos[Z];
pos[Z] = model_info->pos[X];
WTnode_translate(model_info->marso_xform, pos, WTFRAME_PARENT);

/*** set marsokhod heading ***/
WTnode_rotate(model_info->marso_xform, 0.0f, 0.0f, -90.0f + model_info->rot[Z],
        WTFRAME_LOCAL);

WTnode_rotate(model_info->marso_xform, 0.0f, model_info->rot[X], 0.0f,
        WTFRAME_LOCAL);
WTnode_rotate(model_info->marso_xform, model_info->rot[Y], 0.0f, 0.0f,
        WTFRAME_LOCAL);

/*** Front Wheel/Axle Orientation ***/
if (fFunky) {
   /* EASTER EGG - Funky Chicken Dance - progressive Quaternion rotation */
   WTnode_getorientation(model_info->axle_xform[MARSO_FRONT], q);
   WTeuler_2q(radians(model_info->wheel_angle[MARSO_FRONT]), 0.0f, 0.0f, qq);
   WTq_mult(q, qq, q);
   WTeuler_2q(0.0, radians(model_info->axle_angle[FRONT_AXLE]), 0.0, qq);
   WTq_mult(q, qq, q);
   WTnode_setorientation(model_info->axle_xform[MARSO_FRONT], q);
}
else {
   WTeuler_2q(radians(model_info->wheel_angle[MARSO_FRONT]), 0.0f, 0.0f, q);
   WTeuler_2q(0.0, radians(model_info->axle_angle[FRONT_AXLE]), 0.0, qq);
   WTq_mult(q, qq, q);
   WTnode_setorientation(model_info->axle_xform[MARSO_FRONT], q);
}

/*** Middle Wheel/Axle Orientation ***/
WTeuler_2q(radians(model_info->wheel_angle[MARSO_MIDDLE]), 0.0f, 0.0f, q);
WTeuler_2q(0.0, radians(model_info->axle_angle[MIDDLE_AXLE]), 0.0, qq);
WTq_mult(q, qq, q);
WTnode_setorientation(model_info->axle_xform[MARSO_MIDDLE], q);
```

```
	/*** Rear Wheel/Axle Orientation ***/
	WTeuler_2q(radians(model_info->wheel_angle[MARSO_REAR]), 0.0f, 0.0f, q);
	WTeuler_2q(0.0, radians(model_info->axle_angle[REAR_AXLE]), 0.0, qq);
	WTq_mult(q, qq, q);
	WTnode_setorientation(model_info->axle_xform[MARSO_REAR], q);

//	marsokhod_arm_update(marso_node);
}

/**********************************************************************/
/**  Adjust Arm Angles                                              ***/
/**********************************************************************/
void marsokhod_arm_update(WTnode *marso_node)
{
	marsokhod_model_info *info;
	WTq q;

	/*** get internal data structure for marsokhod rover ***/
	info = (marsokhod_model_info *) WTnode_getdata(marso_node);
	WTq_init(q);

	WTnode_setorientation(info->arm_2, q);
	WTnode_setorientation(info->arm_3, q);
	WTnode_setorientation(info->arm_4, q);
	WTnode_setorientation(info->arm_5, q);
	WTnode_setorientation(info->arm_6, q);

	WTnode_rotate(info->arm_2, 0.0f, 0.0f, info->arm_angle[0], WTFRAME_LOCAL);
	WTnode_rotate(info->arm_3, info->arm_angle[1], 0.0f, 0.0f, WTFRAME_LOCAL);
	WTnode_rotate(info->arm_4, info->arm_angle[2], 0.0f, 0.0f, WTFRAME_LOCAL);
	WTnode_rotate(info->arm_5, info->arm_angle[3], 0.0f, 0.0f, WTFRAME_LOCAL);
	WTnode_rotate(info->arm_6, 0.0f, 0.0f, info->arm_angle[4], WTFRAME_LOCAL);
}

/*********************************************************************
***  get elevation of Marsokhod's six wheel pivots using terrain intersection
*********************************************************************/

/** return FALSE if one or more wheels don't intersect terrain **/
/** Always returns true!  This allows an adjustment to be made everytime, **/
/** Fixes previous problem of crashing — CSA **/
char marsokhod_model_elevation(WTnode *marso_node, float height[6])
{
	int wheel, i;
```

```
char test_result = TRUE;
char cNormalize = NO_NORMAL;

WTp3 ray, origin[6], originA[6], originE[6], pVector[6];
float distance;
float distanceA;
float distanceC;
float distanceE;
float fHeight;

WTpoly *intersected_poly;
WTnodepath *np;
WTnodepath *intersected_path;
WTnode *intersected_node;
char *intersected_name;

WTpoly *intersected_polyA;
WTnodepath *npA;
WTnodepath *intersected_pathA;
WTnode *intersected_nodeA;
char *intersected_nameA;

WTpoly *intersected_polyE;
WTnodepath *npE;
WTnodepath *intersected_pathE;
WTnode *intersected_nodeE;
char *intersected_nameE;

marsokhod_model_info *model_info;


WTp3_init(ray);
ray[Y] = 1.0;

/*** get internal data structure for marsokhod rover ***/
model_info = (marsokhod_model_info *) WTnode_getdata(marso_node);

/* raise rover above highest terrain elevation so only test down for intersection */
model_info->pos[Z] = -10.0f;
marsokhod_model_update(marso_node);

/*** Set Line Vectors for Each Wheel ***/
for (wheel = MARSO_FRONT; wheel <= MARSO_REAR; wheel++) {
```

```
        /*** get accumulated translation to Marsokhod's wheel ***/
        np = WTnodepath_new(model_info->wheel_xform[wheel], ims_root, 0);
        WTnodepath_gettranslation(np, origin[wheel]);

        /*** get accumulated translation to Marsokhod's wheel ***/
        np = WTnodepath_new(model_info->wheel_xform[wheel + 3], ims_root, 0);
        WTnodepath_gettranslation(np, origin[wheel + 3]);

        /*** calculate vectors along the axes ***/
        WTp3_subtract(origin[wheel], origin[wheel + 3], pVector[wheel]);
        WTp3_norm(pVector[wheel]);
        WTp3_subtract(origin[wheel + 3], origin[wheel], pVector[wheel + 3]);
        WTp3_norm(pVector[wheel + 3]);

        /*** find additional rayintersect origins ***/
        WTp3_copy(pVector[wheel], originA[wheel]);
        WTp3_mults(originA[wheel], AXLE_TSCALAR);
        WTp3_add(originA[wheel], origin[wheel], originA[wheel]);

        WTp3_copy(pVector[wheel], originE[wheel]);
        WTp3_mults(originE[wheel], -EDGE_TSCALAR);
        WTp3_add(originE[wheel], origin[wheel], originE[wheel]);

        WTp3_copy(pVector[wheel + 3], originA[wheel + 3]);
        WTp3_mults(originA[wheel + 3], AXLE_TSCALAR);
        WTp3_add(originA[wheel + 3], origin[wheel + 3], originA[wheel + 3]);

        WTp3_copy(pVector[wheel + 3], originE[wheel + 3]);
        WTp3_mults(originE[wheel + 3], -EDGE_TSCALAR);
        WTp3_add(originE[wheel + 3], origin[wheel + 3], originE[wheel + 3]);
}

for(wheel = MARSO_LF; wheel <= MARSO_RR; wheel++)  {

        /*** intersect ray with terrain model ***/
        intersected_poly = WTnode_rayintersect(ims_models, ray, origin[wheel],
                &distance, &intersected_path);
        distanceC = distance;
        intersected_polyA = WTnode_rayintersect(ims_models, ray, originA[wheel],
                &distanceA, &intersected_pathA);
        intersected_polyE = WTnode_rayintersect(ims_models, ray, originE[wheel],
                &distanceE, &intersected_pathE);

        if (intersected_poly || intersected_polyA || intersected_polyE) {
```

```
			  if (intersected_poly == NULL) distanceC = 10000.0;
			  if (intersected_polyA == NULL) distanceA = 10000.0;
			  if (intersected_polyE == NULL) distanceE = 10000.0;

			}

			if (intersected_poly) {
//			  height[wheel] = (distance + origin[wheel][Y]);

			  if (fabs(height[wheel] - distance - origin[wheel][Y]) > 0.05f)
			    height[wheel] = height[wheel] + (distance + origin[wheel][Y] -
					height[wheel]) / 4.0f;
			  else height[wheel] = (distance + origin[wheel][Y]);
			}
		}

		return(test_result);
}


/******************************************************************/
/**  Settle Marsokhod Rover Function                            **/
/******************************************************************/

void settle_marso(WTnode *marso_node)
{
		marsokhod_model_info *model_info;
		static float fMid, fAx, fCalc;

		/* absolute wheel elevations numbers front to back, left to right */
		static float WheelElev[6] = {0.0f};

		int axle;

		/** get marsokhod model node data structure with kinematic angles ***/
		model_info = (marsokhod_model_info *) WTnode_getdata(marso_node);

		/** Get height of each wheel pivot above terrain if intersection test success **/
		if(marsokhod_model_elevation(marso_node, WheelElev)) {

			model_info->wheel_angle[MARSO_FRONT] = degrees(-sin((
				WheelElev[MARSO_FRONT + MARSO_LEFT] -
				WheelElev[MARSO_FRONT + MARSO_RIGHT]) /
				MARSO_WHEEL_BASE));
```

```
			model_info->wheel_angle[MARSO_MIDDLE] = degrees(-sin((
				WheelElev[MARSO_MIDDLE + MARSO_LEFT] -
				WheelElev[MARSO_MIDDLE + MARSO_RIGHT]) /
				MARSO_WHEEL_BASE));
			model_info->wheel_angle[MARSO_REAR] = degrees(-sin((
				WheelElev[MARSO_REAR + MARSO_LEFT] -
				WheelElev[MARSO_REAR + MARSO_RIGHT]) /
				MARSO_WHEEL_BASE));
			fMid = (WheelElev[MARSO_MIDDLE + MARSO_LEFT] +
				WheelElev[MARSO_MIDDLE + MARSO_RIGHT]) / 2.0;
			fAx = (WheelElev[MARSO_FRONT + MARSO_LEFT] +
				WheelElev[MARSO_FRONT + MARSO_RIGHT]) / 2.0;
			fCalc = fAx - fMid;
			if (fabs(fCalc) > FB_OFFSET) fCalc = FB_OFFSET * fabs(fCalc) / fCalc;
			model_info->axle_angle[FRONT_AXLE] = degrees(asin((fCalc) / FB_OFFSET));

			fAx = (WheelElev[MARSO_REAR + MARSO_LEFT] +
				WheelElev[MARSO_REAR + MARSO_RIGHT]) / 2.0;
			fCalc = fMid - fAx;
			if (fabs(fCalc) > FB_OFFSET) fCalc = FB_OFFSET * fabs(fCalc) / fCalc;
			model_info->axle_angle[REAR_AXLE] = degrees(asin((fCalc) / FB_OFFSET));

/*** This Calculates the pitch of the Mast and accounts for it in the front and rear axles ***/
			model_info->axle_angle[MIDDLE_AXLE] =
				(model_info->axle_angle[REAR_AXLE] +
				model_info->axle_angle[FRONT_AXLE]) / 2;
			model_info->axle_angle[FRONT_AXLE] -=
				model_info->axle_angle[MIDDLE_AXLE];
			model_info->axle_angle[REAR_AXLE] -=
				model_info->axle_angle[MIDDLE_AXLE];

			/* This compensates for the fact that the Middle Axle is already rotated */
			model_info->wheel_angle[MARSO_FRONT] -=
				model_info->wheel_angle[MARSO_MIDDLE];
			model_info->wheel_angle[MARSO_REAR] -=
				model_info->wheel_angle[MARSO_MIDDLE];

			model_info->pos[Z] = fMid - MARSO_WHEEL_RAD;
		}
		else {
			model_info->pos[Z] = fMid - MARSO_WHEEL_RAD;
		}
}
```