Ashli Crookston
Ashli.Crookston@aggiemail.usu.edu
Work Phone: 435-797-4549

Derek Hampton
Derek.H@aggiemail.usu.edu
Work Phone: 435-797-4549

Rachel Searle
Rachel.Searle@aggiemail.usu.edu
Work Phone: 435-797-4549

March 20, 2009

YangQuan Chen

Department of Electrical Engineering

Utah State University

Dear Dr. Chen:

The following document contains our final design report for Senior Design. It details our design process. This project was assigned to us and sponsored by the Space Dynamics Laboratory's C4ISR (Command, Control, Communications, and Computer Intelligence, Surveillance, and Reconnaissance) Division. The main components of this project were to write unit test code for a critical existing function in the C4ISR Division's code. We set up our code so that it would be compiled automatically using a nightly build and also generate and send error reports to the appropriate developers.

Our mentor for this project was Mr. Pete Krull. If you need to contact Mr. Krull, he can be reached at 435-797-4275 or via email at pete.krull@sdl.usu.edu. Thank you for your time.

Sincerely,

Ashli Crookston

Derek Hampton

Rachel Searle

Final Design Report

Implementation of Automated Testing in a Continuous Integration Development Environment

ECE 4840—Senior Design III
March 20, 2009

Ashli W. Crookston
Derek E. Hampton
Rachel J. Searle

Instructor Approval ______________________________ _____________

Dr. YangQuan Chen Date
Department of Electrical and Computer Engineering
Utah State University

Supervisor Approval ______________________________ _____________

Mr. Pete Krull Date
Software Developer
The Space Dynamics Laboratory

Acknowledgements

We would like to thank the Space Dynamics Laboratory, as well as the Command, Control, Communications, and Computer Intelligence, Surveillance, and Reconnaissance Division for sponsoring our senior project. We would also like to thank our managers and the developers who helped us to perfect and complete our project.

Table of Contents

List of Tables

List of Figures

ABSTRACT

Ashli Crookston, Derek Hampton, and Rachel Searle, per the request of The Space Dynamic Laboratory's Command, Control, Communications, and Computer Intelligence Surveillance, and Reconnaissance Division (C4ISR), designed unit testing software for SDL's image rectification functionality. The unit testing software tests the C4ISR division's existing software for newly introduced errors to the code. This software is automated so that it builds nightly and generates error reports that are sent to the developers. This report will discuss the background and need for our project, a detailed description of the design and implementation of our project, and the results, and also the work breakdown and cost estimates for our project.

1.0 Introduction

Producing and maintaining computer code for a complex program is no easy task. With all of the upgrades and fixes to the code, unrealized errors can be introduced in the overall software program; therefore, the Space Dynamics Laboratory (SDL) hires student testers to perform testing procedures on its software to make sure the software operates correctly. The process of testing the software manually is extremely time-consuming. Doing a complete manual test can take up to two weeks to finish, even if several students are working on it simultaneously. Once the students find the errors or bugs in the code, a lot of time is spent debugging those errors.

Each time a student tester presses a button or selects a menu option, they are essentially executing multiple functions and blocks of code. If an error occurs, the developer looks through the multiple code functions to try and find where the error is being triggered. If the testing can be automated instead, a separate software program would be written to test each individual function within the code itself. If the automated test finds an error, it can report to the developer exactly where the error is occurring. Automatic testing is something that is used in software development to test code on a regular basis and alert developers if any problems arise. This is referred to as "unit testing." The goal of the unit tests is to test every possible input and all limits that are in the code. This provides developers with the knowledge that if something is received that is not expected by the code; there is a case in place that handles the corrupted data that is received. Automating the unit tests supplies a way to handle testing quickly and generates reports for developers indicating whether or not the code is performing as expected.

1.1 Background

According to the company website, "SDL, a unit of the USU Research Foundation, is a nonprofit research corporation owned by Utah State University. Charged with applying basic research to the technology challenges presented in the military and science arenas, SDL has developed revolutionary solutions that are changing the way the world collects and uses data. SDL continues to lead the way in the development of sensors and supporting technologies." [1]

One such technology that SDL develops is an image processing software called "Vantage." Aircraft with sensors capture aerial views of the earth and then the software "allows the image analyst to receive, decompress/compress, process, display, evaluate, exploit, and store imagery data; as well as create/disseminate processed image products. The Vantage Software Suite can also be customized to support data from multiple sensor formats." There are several different types of sensors used, including multi-spectral, hyper-spectral, electro-optical, infrared, and synthetic aperture radar. Many of these sensors are able to see things that aren't visible to the human eye. Figure 1 shows the electromagnetic spectrum. As you can see, only a small portion of the electromagnetic spectrum is visible to the human eye.

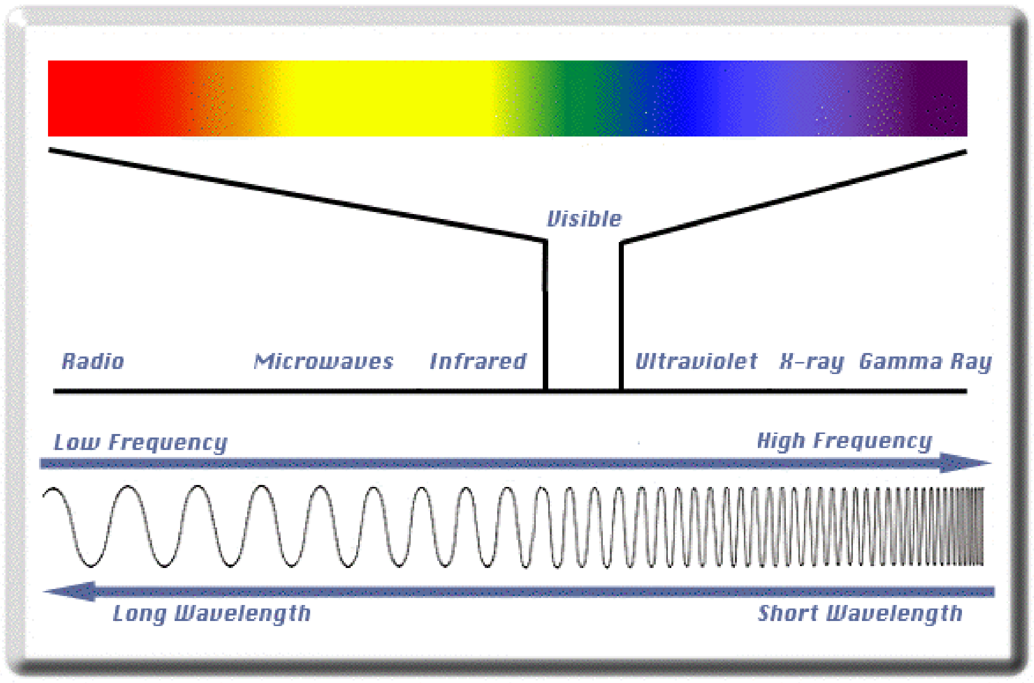


Figure 1: Electromagnetic Spectrum

Figure 2 shows some hyper-spectral imagery. As you can see, the color of the lake makes it appear to be dirty. The reason for this is that with hyper-spectral sensors, different minerals and particles in the water are detected. Even though the human eye cannot see them, hyper-spectral sensors can. When the computer converts the hyper-spectral data into the visible color spectrum, that is, the spectrum that is visible to the human eye, different colors appear which represent the different mineral deposits. There is also a small body of water in the upperright-hand corner of the image. It looks like a different color compared to the larger body of water due to the fact that it has different chemical properties than the large lake, making it give off a different spectrum of color.

Figure 3 shows a large lake bed that has been dried up for some time. To the human eye, it would look like normal desert sand, but after being processed with a multi-spectral camera, the data shows that there is some sort of remaining deposit, possibly vegetation, left over from the old lake.

The greatest benefit of hyper-spectral imaging is its ability to detect things that appear invisible to the human eye. Imagine the possibilities of how this could benefit military efforts in locating objects or sites that are hard to see from high altitudes.

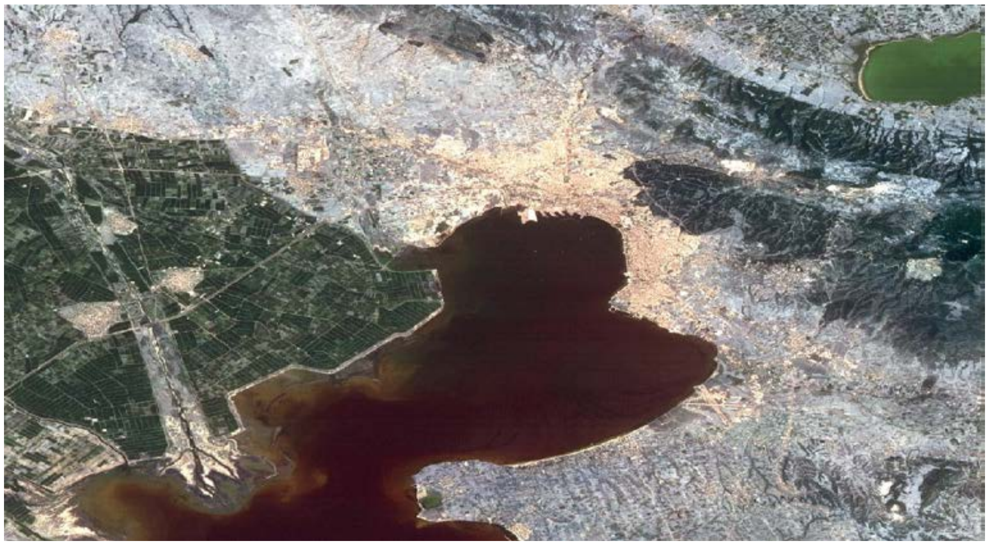


Figure 2: Hyper-spectral Imagery

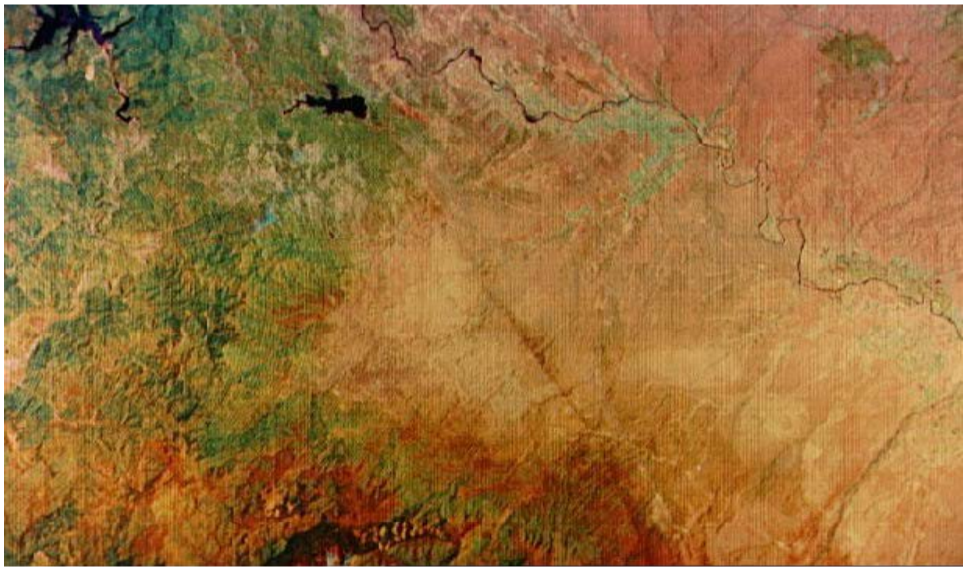


Figure 3: Electro-optical Imagery

There are also several types of imagery collection, including framing, line-scanning, and video. Figure 4 shows the Vantage Screener, which "Displays digital tactical imagery in a robust, near real-time NITF formatted waterfall of decimated imagery from a live data link, solid state recorder, DVD/CD, or hard disk."

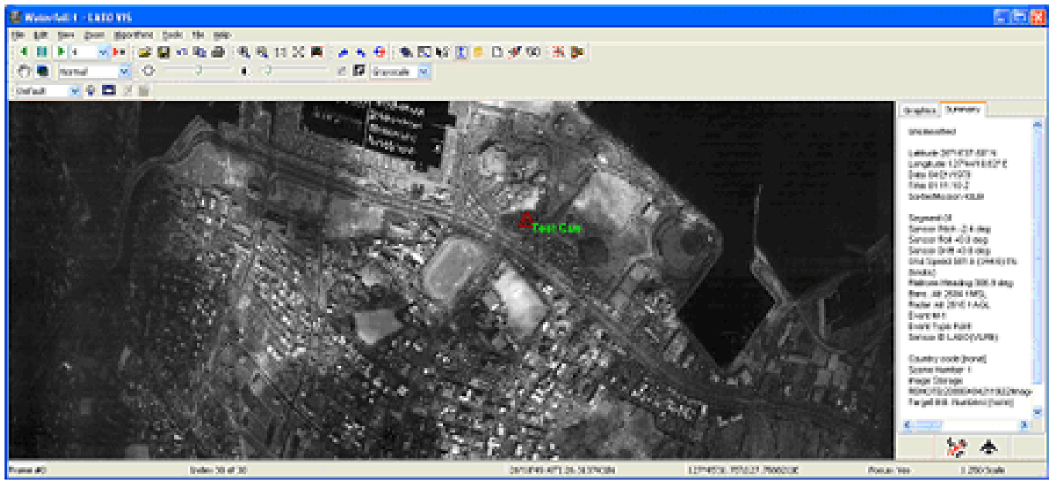
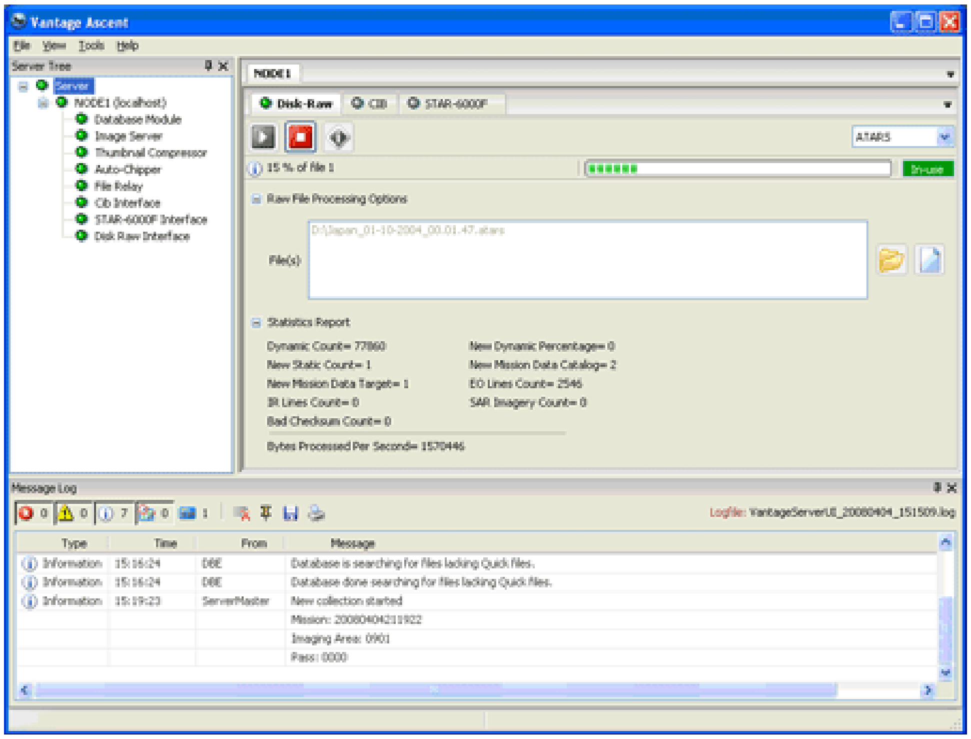


Figure 4: Vantage Screener

Figure 5 shows Vantage Ascent, which acts as a server for Vantage Screener. It “configures ground stations for device interface management (Solid-State Recorders, CDL interface, STANAG 4575, Ethernet feeds, etc.), sensor interface/processing, NITF formation, and database management.”



15 % of file 1

Raw File Processing Options

File(s): D:\Japan_01-10-2004_00.01.47.atars

Statistics Report

```
Dynamic Count= 77860               New Dynamic Percentage= 0
New Static Count= 1                New Mission Data Catalog= 2
New Mission Data Target= 1         EO Lines Count= 2546
IR Lines Count= 0                  SAR Imagery Count= 0
Bad Checksum Count= 0

Bytes Processed Per Second= 1570446
```

Logfile: VantageServerUI_20080404_151509.log

Type	Time	From	Message
Information	15:16:24	DBE	Database is searching for files lacking Quick files.
Information	15:16:24	DBE	Database done searching for files lacking Quick files.
Information	15:19:23	ServerMaster	New collection started
			Mission: 20080404211922
			Imaging Area: 0901
			Pass: 0000

Figure 5: Vantage Ascent

One of the key functions in the Command, Control, Communications, and Computer Intelligence, Surveillance, and Reconnaissance (C4ISR) Division’s Vantage software is the image rectification function. This function is used frequently in many different parts of C4ISR’s code. When the imagery on a user’s screen has the same orientation as it did when the plane flew over and recorded the imagery, it is called “line of flight” data. During image rectification, line-of-flight imagery is adjusted so that North is facing up when the imagery is viewed on a screen. Figure 6 shows line of flight imagery, while Figure 7 shows the same imagery after image rectification (also known as “geo-rectification”).

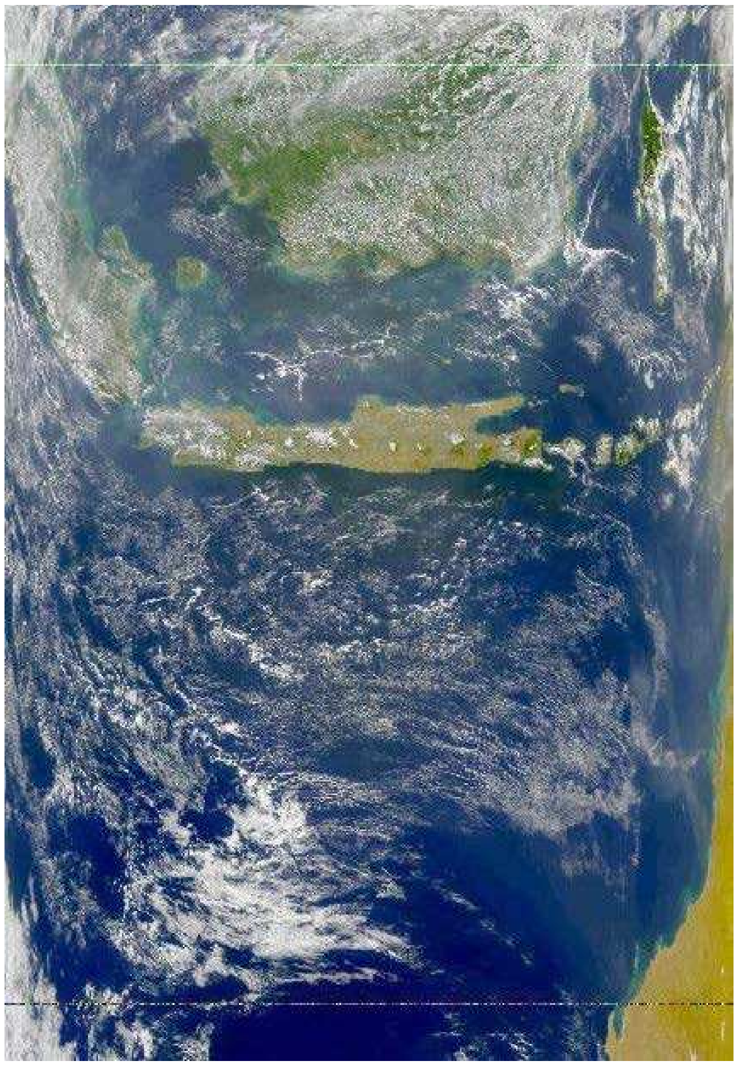


Figure 6: Line of Flight Imagery

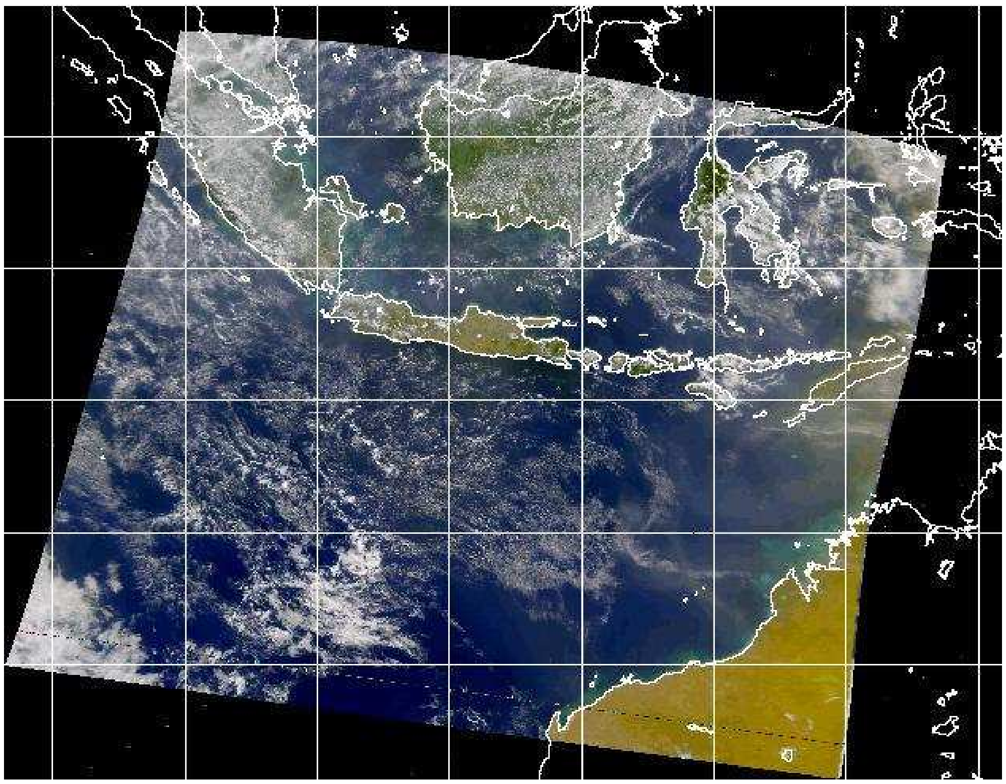


Figure 7: Geo-Rectified Imagery

1.2 Problem Statement

The current testing procedure for the C4ISR division consists mainly of manual testing, which is performed by students, and regression testing, which is performed primarily by full-time testers. The downside of the current testing procedure is that it is extremely time-consuming and there is much opportunity for human errors to overlook certain bugs in the software. Another problem that can occur is when new functionality for C4ISR's software affects old functionality. This happens when improvements to the software cause unwanted results to other parts of the software. Sometimes, these errors are unexpected and are not able to be found in a timely manner. For these reasons, unit testing can be an extremely effective solution to the problem.

1.3 Design Objectives

The main objective of our project is to design and set up test automation that is best-suited for SDL’s C4ISR Division’s software development environment. First, we will need to research different methods for automation and write our tests. Next, we will integrate the unit tests that we have written into the nightly code builds. Once the builds complete, we will need to have the system set up so that developers will be notified via e-mail if their code introduced errors. Finally, this email will provide a general location of where the error occurred.

1.4 Summary of Design Process

Our design process consisted of our technical approach, which involved researching several different methods of unit testing in order to decide which to use to write our code. Next, we held several team meetings and discussed project priorities until we decided on exactly what code to write unit tests for. Originally, we were tasked to work on Project “Save Matrix,” in which we would write unit test code for all of the different types of saves that C4ISR’s software is capable of performing. Priorities changed, however, and we were tasked to work on Project “Image Rectification,” since this is more important to the functionality of C4ISR’s existing code. Once this was decided, we had to familiarize ourselves with the existing code until we knew it well enough to write tests for it. Finally, we wrote the code and then put it through a series of code reviews and edits until it was error-free and fully functional.

1.5 Summary of Final Results

Once our project was completed, the result was code that compiles and works correctly. Our code checks for potential problems created from changes to the image rectification code. Also, once the code is built, e-mails are sent out to the appropriate developer if something they have done has introduced an error to the image rectification code. Our managers and the developers involved in the code reviews are very happy with the project outcome. We are also pleased with what we have been able to accomplish.

1.6 Organization and Summary of Report

The remainder of this report will discuss our preliminary design, including a review of the problem, specifications, and the basic engineering approach, including analysis of decisions that were made. Next, we will discuss our basic solution, design of specific components, and the implementation of our project. Finally, we will discuss cost and assignments before concluding our report.

2.0 Review of Conceptual and Preliminary Design

2.1 Problem Analysis

2.1.1 Review of Problem

The current testing procedure for the C4ISR division mainly consists of manual testing, which is performed by students, and regression testing, which is performed primarily by full-time testers. The downside of the current testing procedure is that it is extremely time-consuming and there are many possibilities for human errors to overlook certain bugs in the software. Another problem that can occur is when new functionality in the C4ISR software effects old functionality. This happens when improvements to the software cause unwanted results to other parts of the software. Sometimes, these errors are unexpected and are not able to be found in a timely manner. For these reasons, unit testing can be an extremely effective solution to the problem.

2.1.2 Summary of Specifications

There were several specifications assigned to our project. First of all, no new software was to be purchased for the project. SDL already had some testing software for us to consider using. There was also a lot of freeware available online for us to use. We were also required to write our unit test code in C++. Finally, once the unit test code was written, we were required to put our code through code reviews so that it would comply with SDL's quality assurance standards.

2.1.3 Discussion of Main Features

The unit testing code has the ability to test every aspect of a function individually within the Image Rectification Project. This allows our software to make sure there are no errors while imagery is being geo-rectified. The unit testing code is added to the build server, which generates and runs the code automatically overnight. If any errors are detected, an email notification is sent to the appropriate developer. This email tells them that they have introduced a bug into the original image rectification code by uploading it to the source repository. Another feature of our unit testing code is that it was written so that the notification email also includes the exact location of the error that has been introduced.

2.1.4 Summary of Basic Engineering Approach

2.1.4.1 Basic Design Concept

The unit testing code is designed to work in coordination with the Boost C++ Libraries. It is to be integrated with SDL's build server that compiles and builds the C4ISR Division's code overnight. The purpose of the unit testing code is to test each part of the Image Rectification Project individually.

Each function in our code is given a test argument. The tested function then returns a value based on the argument that it was given. This returned value is then compared with an expected value. This procedure occurs during the build server's code compilation. If the two values are equal (up to the desired precision of .001), then the test passed. If not, the build fails, and an email is sent to notify the developers that the build failed. The email also informs developers which test function failed, giving the location of the build error origin.

2.2 Decision Analysis

2.2.1 Description of Solution Alternatives

There were several unit testing platforms that were taken into consideration in our design process, including hand-written code, Parasoft's C++Test, and the Boost Test C++ Libraries. We will discuss the pros and cons that we found with each platform and why we made the decision to use the Boost C++ Libraries.

The pros with using purely hand-written code were that we would not have to worry about learning any new software. Additionally, the integration into the nightly builds would work the same way the rest of C4ISR's code is integrated into the nightly builds. The cons of hand-written code, however, were that it is very time-consuming and high-maintenance. The unit test code may also be imperfect, which could cause further errors in the code instead of fixing them.

According to Parasoft's website, "Parasoft C++Test is an integrated solution for automating a broad range of best practices proven to improve software development team productivity and software quality. C++Test enables coding policy enforcement, static analysis, comprehensive code review, and unit and component testing to provide teams a practical way to ensure that their C and C++ code works as expected." The good things about this software were that it generated unit tests automatically. Supposedly, only a few changes and tweaks would have to be made to the auto-generated code in order for the unit testing code to perform as desired. This code is also easily used with C4ISR's auto-build software. Unfortunately, Parasoft made empty promises in our case. We found that the C++Test software did not match up well with SDL's existing software. It also created unit test codes that were unable to compile. Some of them contained thousands of errors that we would have to track down and fix. Parasoft's C++Test turned out to be a very poor platform for our project.

Finally, we considered using the Boost Test C++ Libraries. The good thing about these libraries is that they work well with the C++ standard library and they are easy to use. Many of the developers in the C4ISR division are already using the Boost Libraries to write unit tests for their code, so there were several people who we could turn to if we had any questions. Another great thing about the Boost Test libraries is that they allow for the tests that are produced to run as part of the nightly build. Once the code has been compiled, build errors can be double-clicked in Visual Studio to open the source file and display the exact line of code that failed. The only bad thing about the Boost Test Libraries was that some of the code had to be hand-written, but there were so many built-in unit testing functions that came with these libraries that hand writing the code was trivial. Choosing to use the Boost Test C++ Libraries as the platform for our project was obviously the best decision.

3.0 Basic Solution

Once we were certain that we would be using the Boost Test C++ Libraries to write unit test code for the image rectification functionality, we did a lot of research. We studied the existing Image Rectification code in detail until we understood it very well. Then we decided what functions we would need to write in order to test every aspect of image rectification. These functions include the initialization function as well as five unit testing functions: sensor pitch, sensor roll, sensor heading, aircraft pitch, and aircraft roll, which will be discussed in detail later in this report. Once the code was written, we sent it through code reviews and then integrated it into the nightly build. The nightly build automatically sends out error reports via email to the developers.

3.1 System Components and Relationships

Figure 9 shows a basic block diagram of our system. It contains the original C4ISR code, Boost Test Libraries, unit test code, C++ Compiler, and the build server. The Boost Test Libraries and Original C4ISR code are used to create the unit test code. The integrated code is then sent to the C++ Compiler, which is located on the Build Server, where all of the code is compiled. These components will be discussed in detail later in the report.

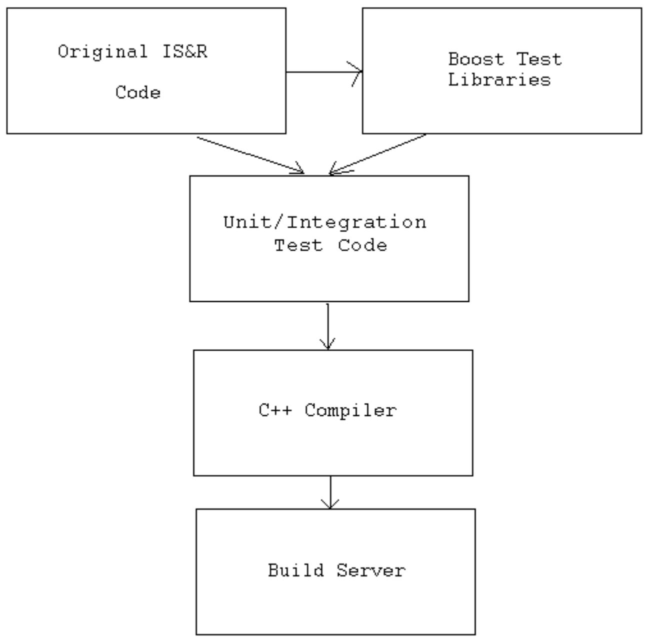


Figure 8: Basic System Block Diagram

3.2 Component-level Specifications

Since the original C4ISR code, Boost Test Libraries, C++ Compiler, and Build Server were already in existence, we did not need to create new specifications for those components. Instead, we had many specifications for our unit test code. First of all, we were required to write our unit test code in C++ and run it through code reviews as mentioned previously. We also had to ensure that our unit test code was compliant with all of the other components in the system. Our unit test code had to be developed based on the original image rectification code and the boost test cases had to be integrated into the unit test code correctly. We also had to ensure that the unit test code could run through the compiler and build server just like the original C4ISR code.

4.0 Performance Optimization and Design of System Components

4.1 Description of Components and Component-level Specifications

The system contains six main components. These components include, the original C4ISR code, the Boost Test Libraries, the unit/integration test code, a C++ Compiler, and a build server.

4.1.1 Original C4ISR Code

The Original C4ISR Code is the software developed by SDL's C4ISR division. The software is called Vantage, and consists of hundreds of thousands of lines of code and dozens of projects. The project that we did our unit testing for was the image rectification project. The Image Rectification project involves Geo-rectifying the imagery so that north is pointing up and lies correctly on a map of the earth below it.

4.1.2 Boost Test Libraries

The Boost Test Libraries are a set of libraries that contain different test functions that can be used with C++ code and using Microsoft Visual Studio. The libraries are freeware found on the internet. We downloaded these libraries and integrated them with the other libraries that are used with the Original C4ISR Code. This enabled the boost libraries to be checked by anyone who checks out the C4ISR Code from the repository and to use the libraries' functionality.

4.1.3 Unit/Integration Test Code

The Unit/Integration Test Code is the code that we wrote to test the Image Rectification functionality. We used the test functions from the Boost Test Libraries to help us test the functionality of Image Rectification. After the unit tests were written and tested, they were integrated into the Original C4ISR Code.

4.1.4 C++ Compiler

The C++ Compiler is what SDL uses to compile the code. We used the C++ Compiler to compile the unit tests for Image Rectification to ensure that the tests compiled without errors. We also used the compiler after we had integrated the tests into the original code to make sure the new tests did not introduce any compile errors to the original code.

4.1.5 Build Server

The Build Server is used at SDL to do nightly builds of the software. It also keeps track of changes made in the code repository and triggers a build upon a change. The Build Server also uses a freeware called CruiseControl to control emails that are sent to developers if a changed they made caused build errors. We use the build server to run our tests on the code every time a build is ran, and if any of the tests failed, developers are notified of the error that occurred during the build.

4.2 Discussion of the Technical Approach Used

Our technical approach involved a lot of research. We researched the different methods that we could use to develop our unit tests. The different methods included handwriting all of the test code, Parasoft's C++ Test, and the Boost Test Libraries. Our research results for the different methods are discussed in section 2.2.1. (We need to link this) Another main aspect to our technical approach was researching the Image Rectification code that we were assigned to write unit tests for. This was a critical part because we had to know exactly what the code was doing in order to test it. Becoming familiar with the Image Rectification code also lead to needing to research roll, pitch, and yaw angles; this ended up being a very complicated and a key part to our unit tests.

4.3 Discussion of Design Details

After all of our research was completed, we began to write our unit test code for image rectification. There were six tests and functions that we wrote to test image rectification. These tests were, the initialization function, sensor pitch, heading (yaw), and roll, and aircraft pitch and roll. We also tested aircraft heading, but the tests were built into the other tests.

In order to make the correct geo-rectification calculations, we took several rotation types into account. These are roll, pitch, and heading, which is sometimes called "yaw." They will be discussed in further detail in Section 4. Roll is the airplane or sensor's rotation around the z-axis, pitch is the rotation around the x-axis, and heading is rotation around the y-axis. Figure 8 shows a diagram of roll, pitch, and yaw with respect to an airplane.

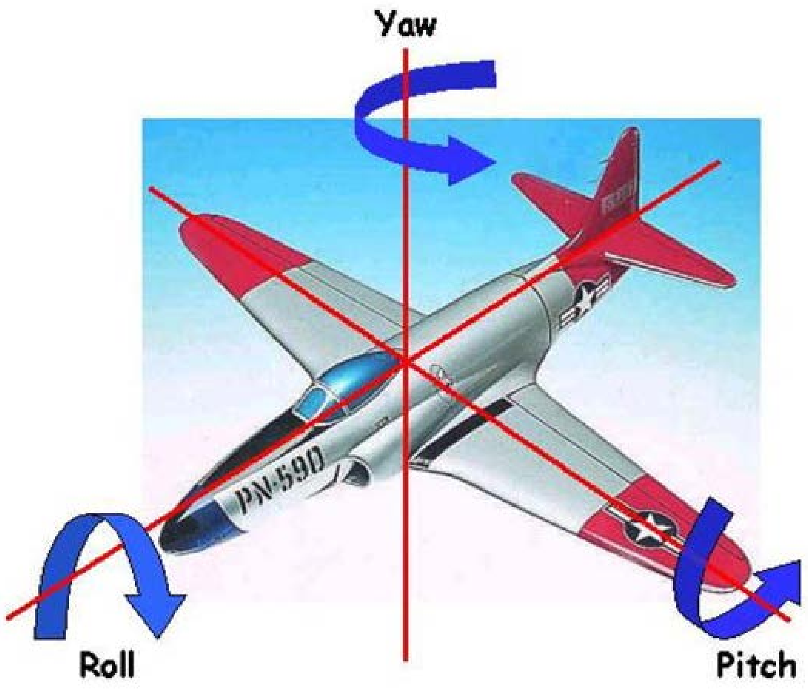


Figure 9: Roll, Pitch, and Yaw Diagram

4.3.1 Initialization

The initialization function is where we made any initial calculations and initialized any variables. Most of the initialization that was needed was to enable the function call to the Image Rectification code we were testing. Most of the values did not affect our test but were required to be initialized to make the function call. Other variables that we initialized in this function did affect our testing and had to be calculated correctly to make the test results correct. Part of our research in the technical approach involved figuring out these calculations but we also verified with the developer over the Image Rectification that we were performing the calculations correctly.

4.3.2 Sensor Pitch

Pitch is the angle that tells you at what degree the sensor is pointing toward the earth. For example, if the sensor pitch is zero degrees, the sensor is pointing out toward the front of the plane, and would not see much of the earth. At a pitch of negative 90 degrees, the sensor is pointing directly down at the earth. At this angle the sensor direction is perpendicular to the earth's surface. If the sensor pitch was -180 degrees, the sensor would be at the same angle to the earth if the pitch were 0 degrees, except it would be facing toward the rear of the plane instead of the front. So, to test the sensor pitch, we changed the varied the pitch angle from the original setting, and verified that the ground latitude and longitude of where the sensor pointed to changed correctly based on the pitch change.

4.3.3 Sensor Heading

The sensor heading or yaw is the angle that determines to what degree the sensor is pointing north, south, east or west. For instance, a zero degree heading means it is pointing north, 45 degrees is east, negative 45 degrees is west, and 180 and negative 180 is south. However, with the sensor, the heading is relative to the airplane. This means that if the aircraft is flying straight east, having a yaw of 45 degrees, the sensor's yaw, at zero, would be aligned with the plane also heading east. Testing this angle was particularly complicated because of the relativity to the airplane. But the test idea is similar to the sensor pitch. So, we changed the sensor heading from the original setting and verified that the ground latitude and longitude changed accordingly. One thing we ran into that also complicated the testing for sensor heading was for the case that the sensor pitch is -90 degrees. Through our research of the roll, pitch, and yaw angles, we found that with a pitch of -90 degrees the ground latitude and longitude doesn't change when the heading is changed. So, when we wrote our tests, we had to account for this specific case.

4.3.4 Sensor Roll

The way the image rectification code uses the roll, pitch, and yaw angles, the sensor roll does not affect the ground latitude and longitude. This is not true for all applications though. It just happens that the C4ISR software has been written so that this angle can be neglected. However, even though this angle is neglected we still had to test for every type of situation to make sure that the code responded as expected for a change in sensor roll, meaning the ground latitude and longitude does not change.

4.3.5 Aircraft Pitch

The aircraft pitch test is similar to the sensor pitch test. However, the aircraft is allowed to have a slight positive pitch, unlike the sensor, and so tests were added to account for a slight positive pitch of the aircraft. Also, through our research and discussion with developers we made decisions that the aircraft would never be pointing straight up or down, or trying to flip over, so the pitch range for the aircraft that we tested was between -90 and 90 degrees but never was less than or equal to -90 or greater than or equal to 90 degrees.

4.3.6 Aircraft Roll

Unlike sensor roll, aircraft roll had a big affect on the ground latitude and longitude. This is because the ground latitude and longitude is based on the orientation of the sensor, so if the aircraft is tilted or rolled slight left or right, it will greatly change the point on the ground that the sensor is pointing to. This test was a very complicated test, because how the ground latitude and longitude changed with different roll angles is dependent on both the sensor heading and aircraft heading. So, we had to go through many calculations to make sure that we were correctly checking the ground latitude and longitude when the aircraft roll was changed.

4.3.7 Aircraft Heading

The tests for the aircraft heading were different than the other tests. This was because we incorporated the aircraft heading tests with the other tests. What this entailed was just changing the aircraft heading alongside the other angles that needed testing. The aircraft heading is similar to what was described above for sensor heading, except that if the heading is zero degrees, then the aircraft is flying directly north. We tested nine different headings for the aircraft. These headings were, 0, +45 and – 45, +90 and -90, +135 and -135, and +180 and -180 degrees. For example, at each heading we tested every sensor pitch angle. This was accomplished fairly easily using a nested for loop, but made verifying the ground latitude and longitude very difficult.

4.3.8 Integration

Once our image rectification unit tests were written, our code underwent a detailed code review by two of the C4ISR division's full-time software developers. Our first code reviewer advised two pages worth of corrections for us to make to our code. Most of these corrections were simple, stylistic changes. Once we made those changes, he reviewed our code a second time and then approved it. After that, we passed our code on to the second code reviewer. He suggested another two pages worth of corrections, and these changes were more critical. First, he required a major change in the structure of our code. Originally, we wrote our code in a functional style, but this developer required that our code be changed to an object-oriented style. The reasons for this were that it made our code easier to maintain and also improved its readability. In order to do this, we had to create a separate header file and organize our data into classes and structs. Once these changes were made, the second developer reviewed our code again and suggested many stylistic changes. After five total code reviews between these two developers, our code was greatly improved. It was much more efficient and the layout was nicer. The code reviews helped us to better learn how to develop software in a professional environment. They were critical to our project, and extremely useful for our professional experience.

4.4 Engineering Drawings and Schematics

The overall unit system for our project consists of the original C4ISR code, the Boost Test Libraries, our unit testing code, a C++ compiler, and a build server. The original C4ISR code is currently being developed at SDL and is what we are writing the unit tests for. It has already been developed by the C4ISR division at SDL. It consists of hundreds of thousands of lines of code and dozens of projects. The Boost Test Libraries are the platform that we used to write our unit test code with. The unit testing code is code written to test specific, existing C4ISR code thoroughly by exercising all necessary functions with many different values. The build server is dedicated to nightly builds of the C4ISR division's software, including our unit testing code. It also tracks changes in the repository and updates upon a change in the code. Figure 10 shows the block diagram for our entire project.

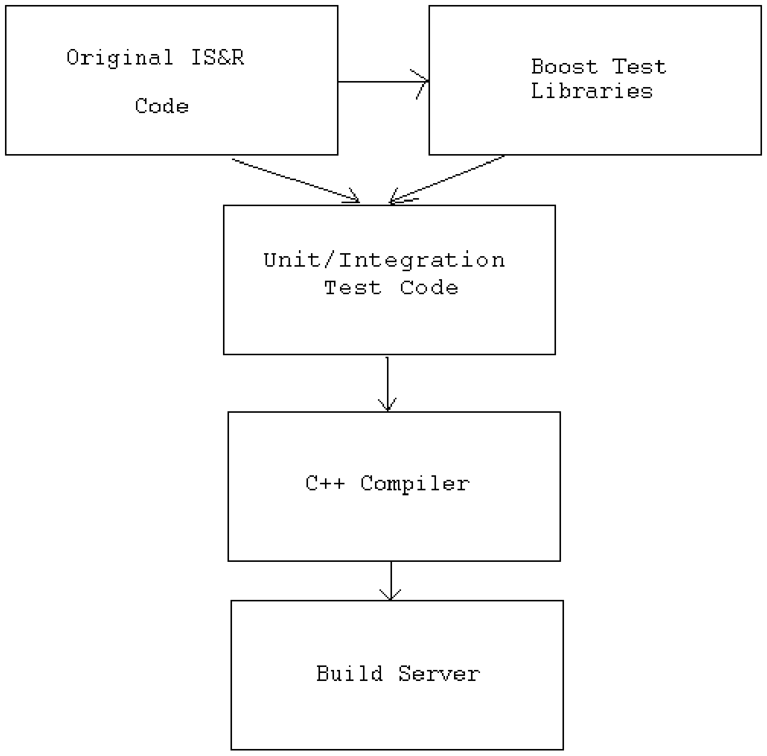


Figure 10: Basic System Block Diagram

The integrated code is built using a C++ compiler on the build server. Automated builds are completed using a program called, "CruiseControl." Builds are triggered every night ("nightly build"). Builds are also triggered upon a change made in the code repository. During this build, error reports are generated and sent out to the developers. These reports let the developers know if they have introduced a bug to the code, and if so, where that bug is located.

4.5 Summary of Final Design Results

After going through the multiple code reviews, the final design of our code was extremely improved. Our final revision of the unit testing code cut out about 2000 lines of code from the original first draft and made our code much more efficient and readable. Our manager and developers that we worked with on the design of our code are extremely satisfied with our unit tests and their performance.

4.6 Performance Evaluation

Our code was tested extensively to verify correct functionality. The unit tests were introduced into the original C4ISR code without causing any errors. If there is an error in the image rectification code, meaningful error messages are displayed and the developers are notified that they introduced an error into the image rectification code. The developers are satisfied with the unit testing for image rectification.

5.0 Project Implementation, Operation, and Assessment

5.1 Details of Implementation

Once the unit test code was written and perfected, it was time to integrate it into the system used by C4ISR. Figure 11 is a diagram of how the system works.

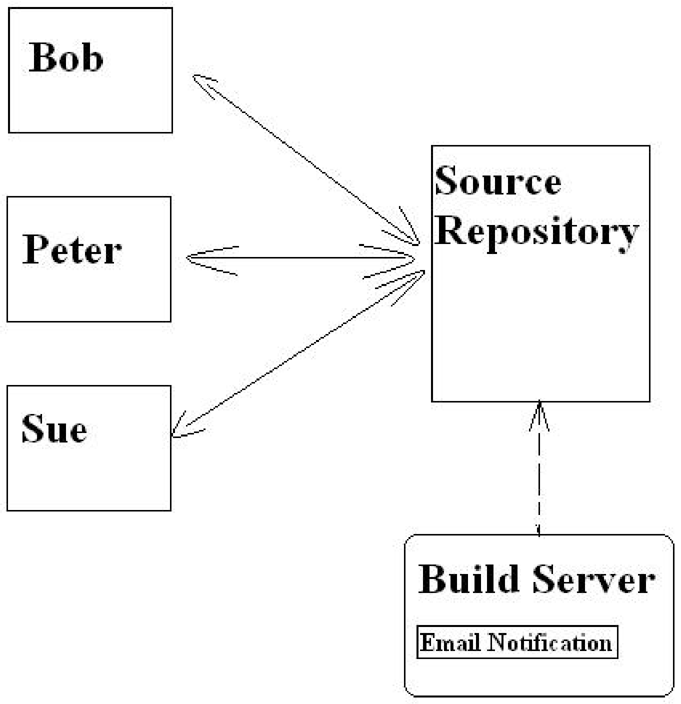


Figure 11: Build System Diagram

It starts out with the developers, in this case Bob, Peter, and Sue. For example, Bob just finished writing some code and now he wants to integrate it into the system. First, he would check his changes into the source repository. The source repository is where all of C4ISR's code is stored. All of the developers have access to the source repository, so they can access the code there by "checking it out." The build server is notified every time that code is checked into the source repository, which means that there has been a change to the code. The Build Server then compiles the code. The Build Server also compiles the code during the "nightly build," so that the students and developers can obtain a fresh build of the code every morning when they come into work. When the Build Server finishes compiling (or an attempt at compiling), if any errors were encountered, an email notification is sent out to the developers, or in our case, if our unit testing code finds a bug that was introduced to the image rectification code, an email notification is also sent out to the developers. These email notifications contain the exact location of the error or bug so that the developers can find and fix it in a timely fashion.

5.2 Operational Test Results

Our Unit Test code was integrated into the system successfully. Once it was checked into the source repository, a build was triggered on the Build Server and it compiled successfully. Email notifications are sent out to developers if a bug has been introduced to the original image rectification code. Our project works perfectly with the rest of the system. We did not need to make any changes based on our design and testing results.

6.0 Final Scope of Work Statement

6.1 Summarize What has Been Done

At this point, our project is complete. We started our project by researching unit testing, including different methods of unit testing and various platforms that we could use for our testing. We started work on the "Save Matrix" Unit Testing Project, but through discussions with our manager, determined that the "Image Rectification" Unit Testing Project was higher priority, so we re-directed our focus. We researched image rectification, including how it works and the math behind it. We became very familiar with the existing image rectification code so that we could write the best unit tests possible for it. We completed our code and then refined and perfected it through code reviews. Next our code was integrated into the nightly build and set up so that error reports would go out to any necessary developers. Our project was a success and will have a great impact on the C4ISR Division's code.

6.2 Lessons Learned and Suggestions for Future Activities

Throughout the course of this project, we learned some very important principles of software engineering. First of all, too many function blocks within the same file can make it difficult for others to understand the functionality of our code. Using C++ classes helps to organize similar function blocks into more understandable events that others can follow. Good code is created by understanding and reviewing the desired purposes and goals that the code will accomplish and then writing code according to those goals. Reviewing the code on your own and having others review it via code reviews are critical elements to software design. It is important to fine-tune the code until it is as efficient and readable as possible. Chances are good that someone else will have to read, understand, and then edit your code someday, so they will greatly appreciate this.

6.2 Related Project Management Issues

The unit testing code that was created is used for only one of the several projects involved in SDL's software. The unit test project demonstrates that software can test other software more efficiently and effectively than manual regression testing by a human can. SDL is now transitioning to asking student employees to write unit test code instead of having them do purely manual testing. This will allow SDL to have a better testing procedure since unit testing is more efficient and effective than manual testing.

7.0 Other Issues

7.1 Component Suppliers

All of the components for our project were supplied by the Space Dynamics Laboratory.

7.2 Reliability

Our unit test code has been tested extensively. The compilation results show that the code is reliable and error-free. In case of an error in the original code, the unit test code performs the desired results, meaning it detects the error and emails the software developers with the location of the error so that it can be fixed as quickly as possible.

7.3 Global, Economic, and Social Impact

Our unit testing software has a global impact because it is perfecting software that can and has saved many lives on both sides of war efforts. Automated testing will also produce higher quality software which will be used in future Department of Defense efforts.

The societal impact of our project is that those working at SDL will be able to spend less time debugging (which is often boring and monotonous) and more time enhancing the software, which is typically a more enjoyable part of any software development job.

The economic impact of our project is that automated testing will allow SDL to produce a better product. This will increase customer confidence. The developers will be able to use their time more efficiently because they will be able to focus on spending more time enhancing the software than trying to track down pesky bugs. It is also commonly known that the longer a bug goes undetected, the more expensive it is to fix. Our code will catch errors in the code the same day that they were created.

7.4 Maintenance

Because of the style and commenting of our unit testing code, it will be easy for C4ISR's software developers to maintain. Our code flows well, it is object-oriented, and it is filled with detailed comments. A specific developer ahs been assigned to maintain our code, however, no immediate maintenance issues are foreseen.

7.5 Contracts and other legal/ethical Issues

Our code is owned by the C4ISR Division at the Space Dynamics Laboratory. Our code is proprietary; therefore it may not be released to the public.

7.6 Product Documentation

Since our project did not change the actual functionality of C4ISR's software, no new documentation or changes in SDL's software documentation were required. Our project was well documented by following the guidelines given to us for the senior design course. We also took care of documentation for our project by writing very detailed, well-commented code.

7.7 Operating Procedures

The operation of our system is extremely simple. Since our code is integrated into the build server, as soon as it was completed and checked in, it runs automatically. The error reports sent to the developers are also generated and sent out automatically.

7.8 Contractor and Supervision

The contractor for our project was the Space Dynamics Laboratory. We worked under the supervision of Mr. Pete Krull, Control Manager of the C4ISR Division.

7.9 Inspection

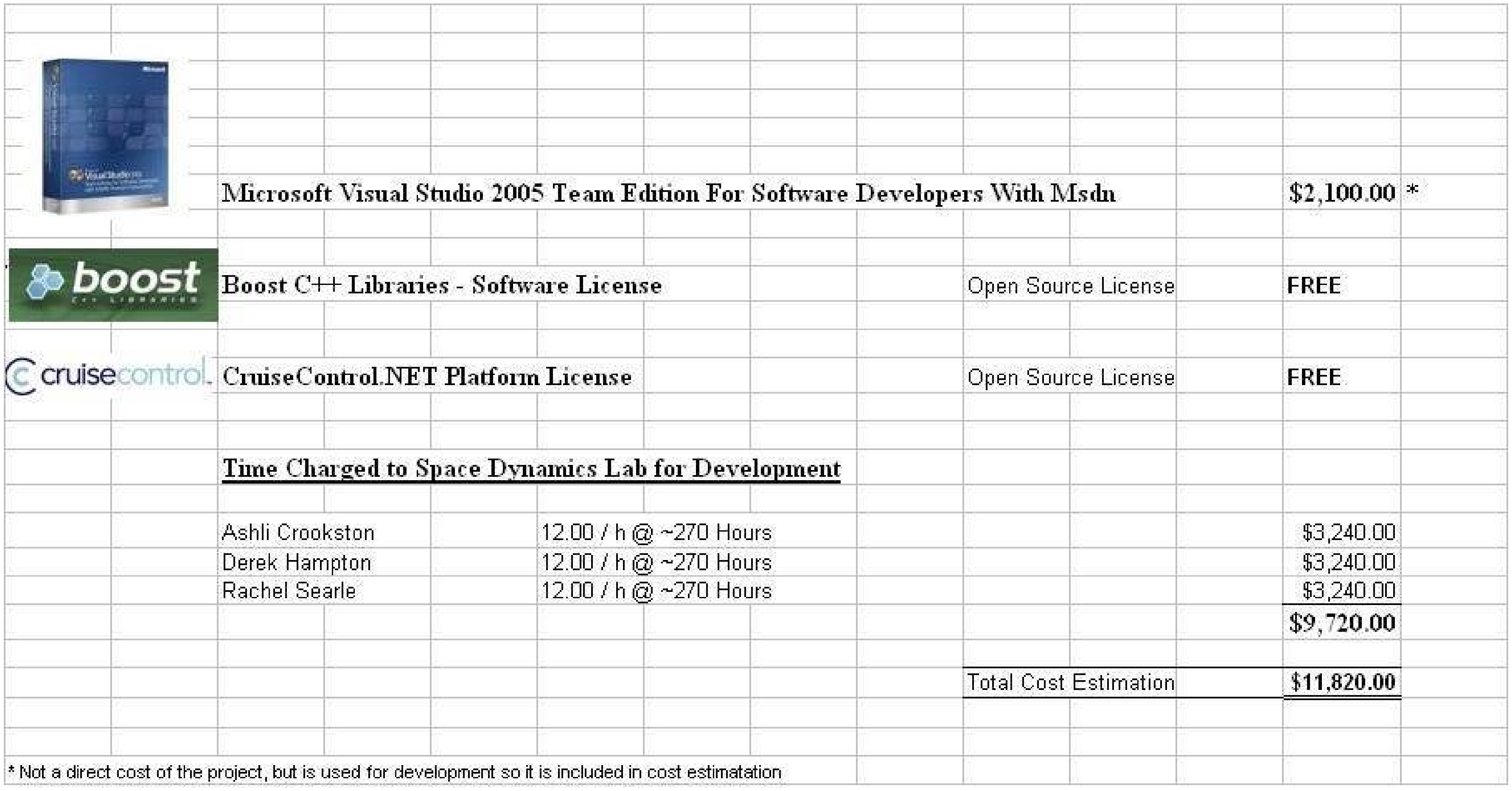
Our software was inspected via code reviewers. In the future, it will be inspected by the developer assigned to the original image rectification code.

7.10 Quality assurance

To ensure that our code met SDL's quality assurance standards, we were required to make sure that the format of the introduced code was in harmony with the previously existing code. We also had to ensure that the new code was necessary, efficient, readable, and understandable. The code reviews helped us to meet all of these requirements.

8.0 Cost Estimation

Table 1 shows a breakdown of the costs for our project. We used Microsoft Visual Studio 2005 to develop our code. A license for this product typically costs $2100, but since SDL already owned licenses for this product prior to our commencement on our senior project, we do not consider it to be a direct cost to our project. We also used the Boost C++Libraries and CruiseControl.NET to create our project, but both of these products have open source licenses, so they are free to use. The only expense for our project was the man hours charged to the Space Dynamics Laboratory for the development of our unit test code. We estimated that this cost was just under $10,000.



Microsoft Visual Studio 2005 Team Edition For Software Developers With Msdn			$2,100.00	*
Boost C++ Libraries - Software License		Open Source License	FREE	
CruiseControl.NET Platform License		Open Source License	FREE	
<u>**Time Charged to Space Dynamics Lab for Development**</u>				
Ashli Crookston	12.00 / h @ ~270 Hours		$3,240.00	
Derek Hampton	12.00 / h @ ~270 Hours		$3,240.00	
Rachel Searle	12.00 / h @ ~270 Hours		$3,240.00	
			$9,720.00	
		Total Cost Estimation	**$11,820.00**	

* Not a direct cost of the project, but is used for development so it is included in cost estimatation

Table 1: Cost Estimation

9.0 Project Management Summary

9.1 Tasks: What Has Been Done

*Research on unit testing

*Research on possible unit testing platforms

*Unit tests auto-generated by Parasoft C++Test

*Correction of auto-generated unit tests

*Familiarization with Boost Test Libraries

*Refocus of project to work on Image Rectification

*Write unit test code

*Major reformatting of code

*Code reviews

*Editing of code

*Integration of code into the system

*Testing our project

*Completion of project

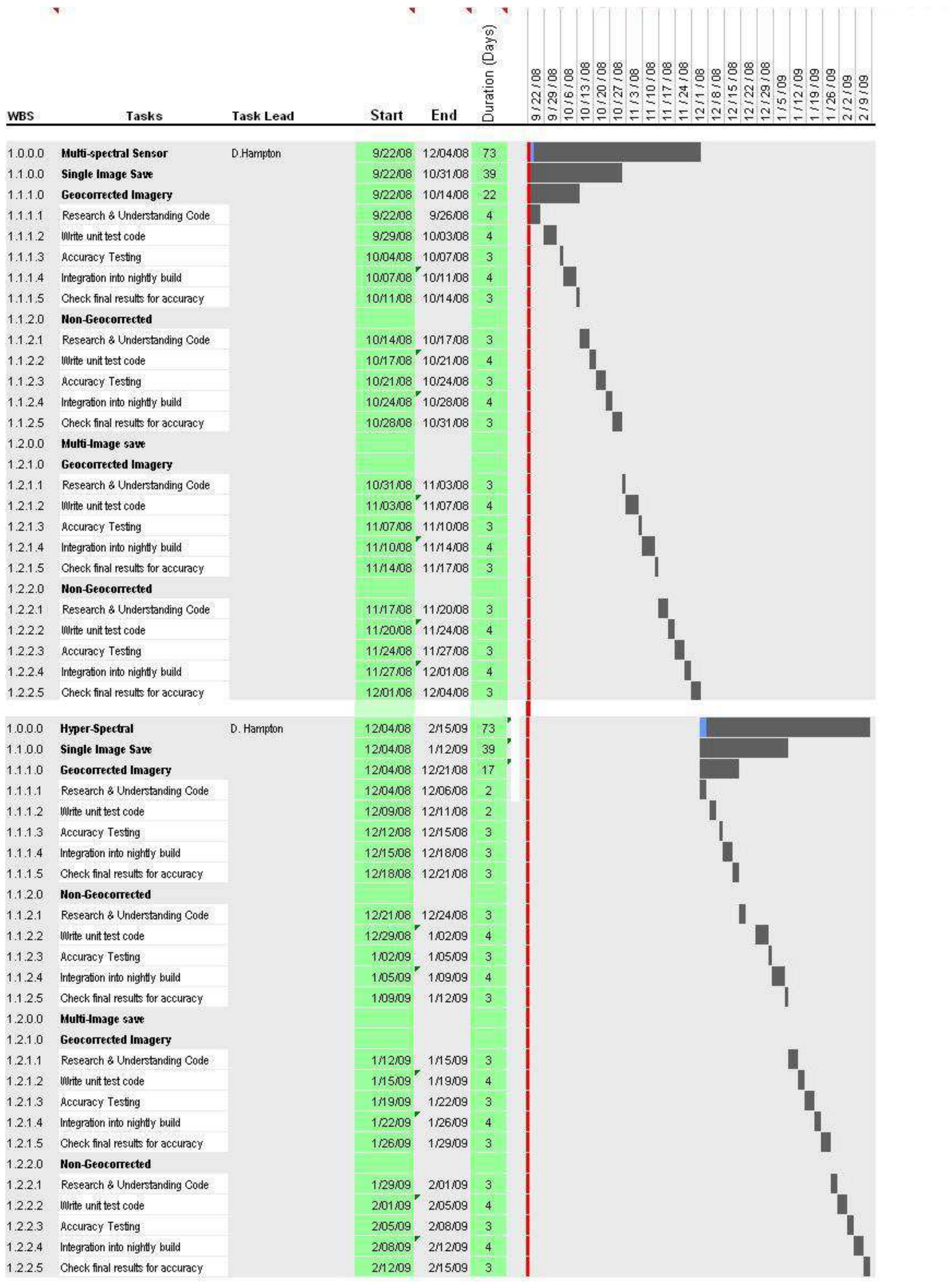
9.2 Tasks: What still needs to be done

*Basic upkeep of code

*Some documentation

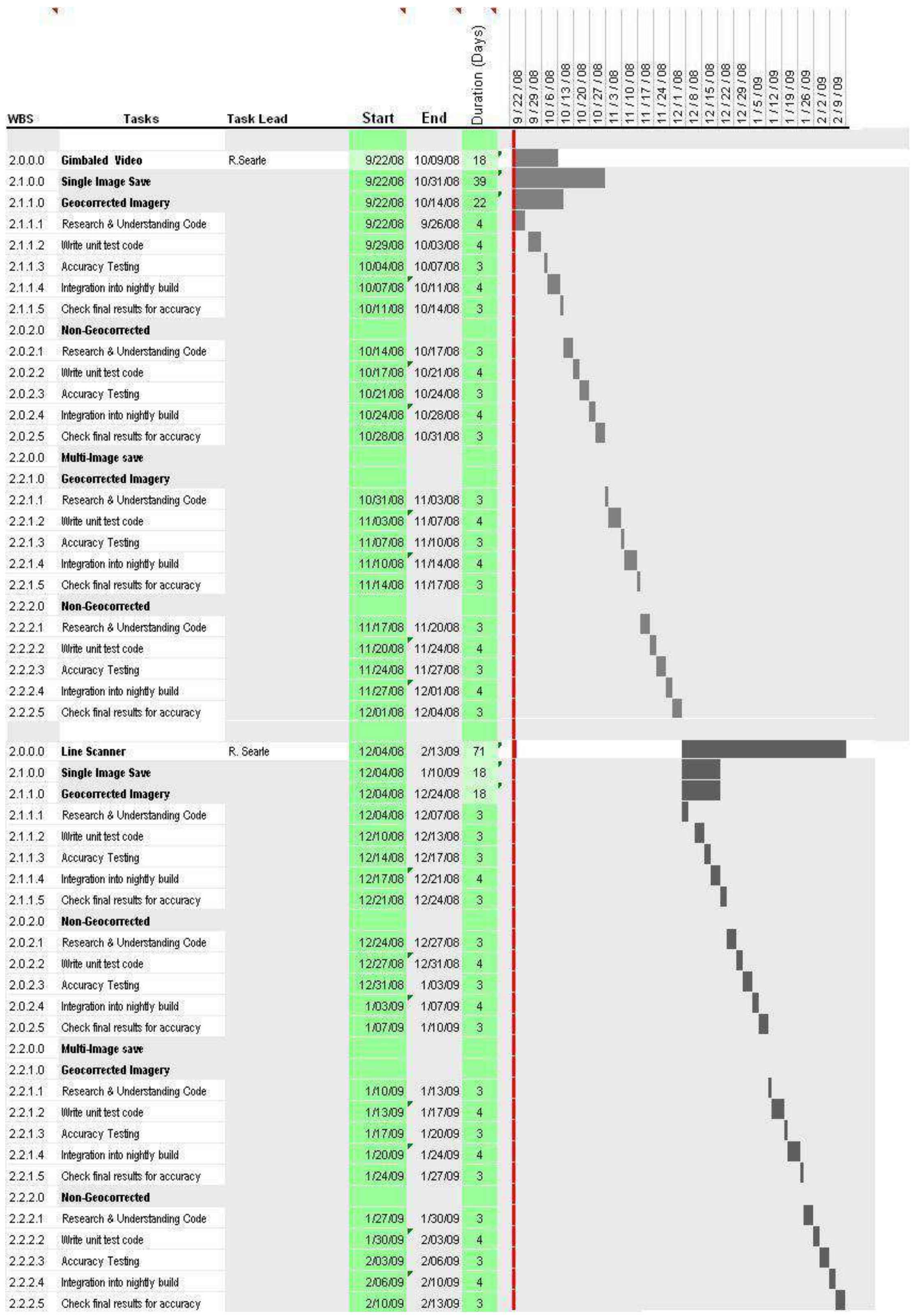
9.3 Time: Gantt Chart

The next four figures are Gantt charts, which show the activities that were completed for our project, as well as the duration and the order of each of these activities. The first three (Figures 12, 13, and 14) Gantt Charts cover the original plans for our Save Matrix Unit Testing project. The fourth Gantt chart (Figure 15) shows everything from October 27th (when our project changed to Image Rectification Unit Testing) until the completion of the project.



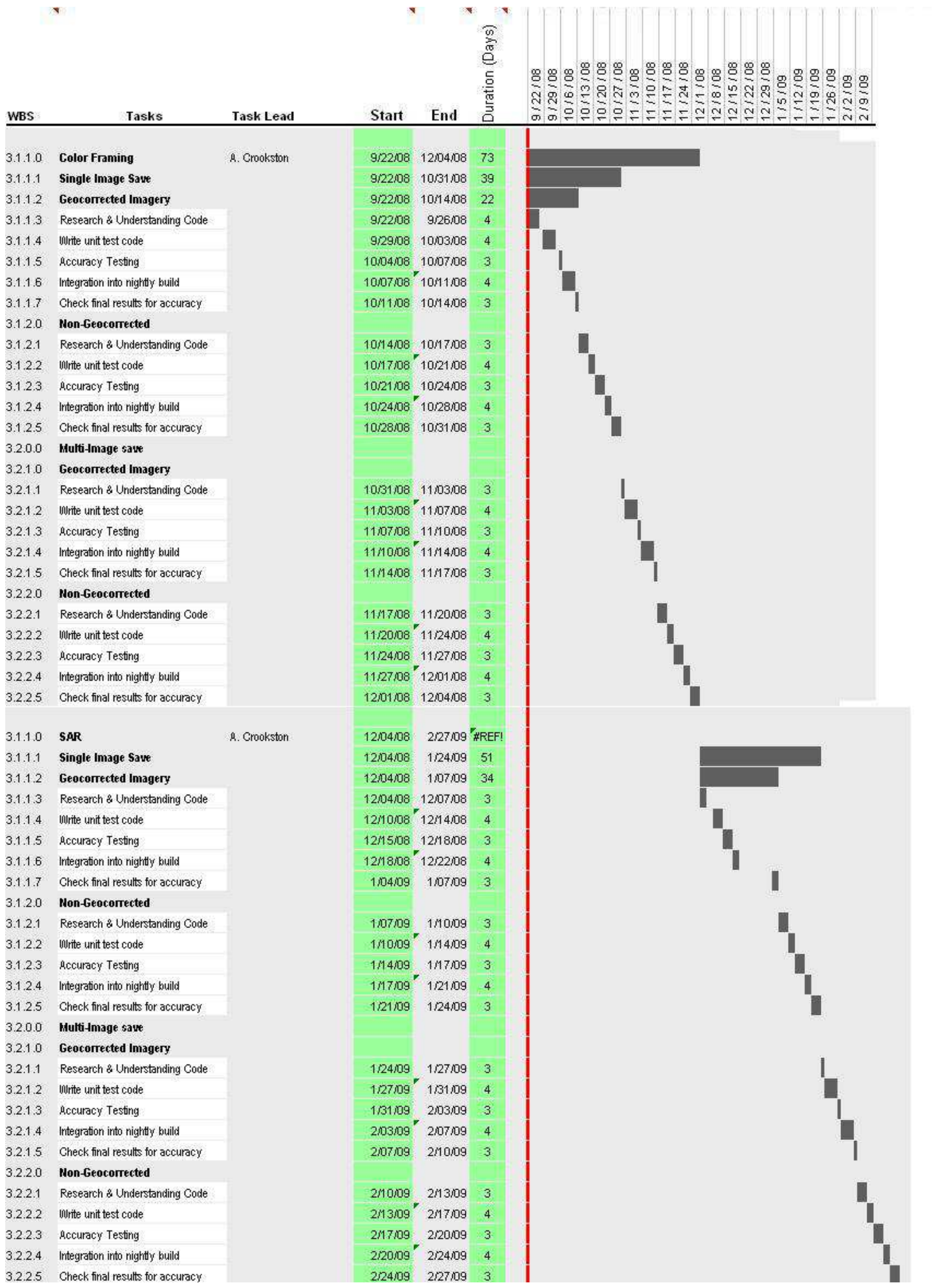
WBS	Tasks	Task Lead	Start	End	Duration (Days)
1.0.0.0	**Multi-spectral Sensor**	D.Hampton	9/22/08	12/04/08	73
1.1.0.0	**Single Image Save**		9/22/08	10/31/08	39
1.1.1.0	**Geocorrected Imagery**		9/22/08	10/14/08	22
1.1.1.1	Research & Understanding Code		9/22/08	9/26/08	4
1.1.1.2	Write unit test code		9/29/08	10/03/08	4
1.1.1.3	Accuracy Testing		10/04/08	10/07/08	3
1.1.1.4	Integration into nightly build		10/07/08	10/11/08	4
1.1.1.5	Check final results for accuracy		10/11/08	10/14/08	3
1.1.2.0	**Non-Geocorrected**				
1.1.2.1	Research & Understanding Code		10/14/08	10/17/08	3
1.1.2.2	Write unit test code		10/17/08	10/21/08	4
1.1.2.3	Accuracy Testing		10/21/08	10/24/08	3
1.1.2.4	Integration into nightly build		10/24/08	10/28/08	4
1.1.2.5	Check final results for accuracy		10/28/08	10/31/08	3
1.2.0.0	**Multi-Image save**				
1.2.1.0	**Geocorrected Imagery**				
1.2.1.1	Research & Understanding Code		10/31/08	11/03/08	3
1.2.1.2	Write unit test code		11/03/08	11/07/08	4
1.2.1.3	Accuracy Testing		11/07/08	11/10/08	3
1.2.1.4	Integration into nightly build		11/10/08	11/14/08	4
1.2.1.5	Check final results for accuracy		11/14/08	11/17/08	3
1.2.2.0	**Non-Geocorrected**				
1.2.2.1	Research & Understanding Code		11/17/08	11/20/08	3
1.2.2.2	Write unit test code		11/20/08	11/24/08	4
1.2.2.3	Accuracy Testing		11/24/08	11/27/08	3
1.2.2.4	Integration into nightly build		11/27/08	12/01/08	4
1.2.2.5	Check final results for accuracy		12/01/08	12/04/08	3
1.0.0.0	**Hyper-Spectral**	D. Hampton	12/04/08	2/15/09	73
1.1.0.0	**Single Image Save**		12/04/08	1/12/09	39
1.1.1.0	**Geocorrected Imagery**		12/04/08	12/21/08	17
1.1.1.1	Research & Understanding Code		12/04/08	12/06/08	2
1.1.1.2	Write unit test code		12/09/08	12/11/08	2
1.1.1.3	Accuracy Testing		12/12/08	12/15/08	3
1.1.1.4	Integration into nightly build		12/15/08	12/18/08	3
1.1.1.5	Check final results for accuracy		12/18/08	12/21/08	3
1.1.2.0	**Non-Geocorrected**				
1.1.2.1	Research & Understanding Code		12/21/08	12/24/08	3
1.1.2.2	Write unit test code		12/29/08	1/02/09	4
1.1.2.3	Accuracy Testing		1/02/09	1/05/09	3
1.1.2.4	Integration into nightly build		1/05/09	1/09/09	4
1.1.2.5	Check final results for accuracy		1/09/09	1/12/09	3
1.2.0.0	**Multi-Image save**				
1.2.1.0	**Geocorrected Imagery**				
1.2.1.1	Research & Understanding Code		1/12/09	1/15/09	3
1.2.1.2	Write unit test code		1/15/09	1/19/09	4
1.2.1.3	Accuracy Testing		1/19/09	1/22/09	3
1.2.1.4	Integration into nightly build		1/22/09	1/26/09	4
1.2.1.5	Check final results for accuracy		1/26/09	1/29/09	3
1.2.2.0	**Non-Geocorrected**				
1.2.2.1	Research & Understanding Code		1/29/09	2/01/09	3
1.2.2.2	Write unit test code		2/01/09	2/05/09	4
1.2.2.3	Accuracy Testing		2/05/09	2/08/09	3
1.2.2.4	Integration into nightly build		2/08/09	2/12/09	4
1.2.2.5	Check final results for accuracy		2/12/09	2/15/09	3

Figure 12: Gantt Chart 1



WBS	Tasks	Task Lead	Start	End	Duration (Days)
2.0.0.0	**Gimbaled Video**	R.Searle	9/22/08	10/09/08	18
2.1.0.0	**Single Image Save**		9/22/08	10/31/08	39
2.1.1.0	**Geocorrected Imagery**		9/22/08	10/14/08	22
2.1.1.1	Research & Understanding Code		9/22/08	9/26/08	4
2.1.1.2	Write unit test code		9/29/08	10/03/08	4
2.1.1.3	Accuracy Testing		10/04/08	10/07/08	3
2.1.1.4	Integration into nightly build		10/07/08	10/11/08	4
2.1.1.5	Check final results for accuracy		10/11/08	10/14/08	3
2.0.2.0	**Non-Geocorrected**				
2.0.2.1	Research & Understanding Code		10/14/08	10/17/08	3
2.0.2.2	Write unit test code		10/17/08	10/21/08	4
2.0.2.3	Accuracy Testing		10/21/08	10/24/08	3
2.0.2.4	Integration into nightly build		10/24/08	10/28/08	4
2.0.2.5	Check final results for accuracy		10/28/08	10/31/08	3
2.2.0.0	**Multi-Image save**				
2.2.1.0	**Geocorrected Imagery**				
2.2.1.1	Research & Understanding Code		10/31/08	11/03/08	3
2.2.1.2	Write unit test code		11/03/08	11/07/08	4
2.2.1.3	Accuracy Testing		11/07/08	11/10/08	3
2.2.1.4	Integration into nightly build		11/10/08	11/14/08	4
2.2.1.5	Check final results for accuracy		11/14/08	11/17/08	3
2.2.2.0	**Non-Geocorrected**				
2.2.2.1	Research & Understanding Code		11/17/08	11/20/08	3
2.2.2.2	Write unit test code		11/20/08	11/24/08	4
2.2.2.3	Accuracy Testing		11/24/08	11/27/08	3
2.2.2.4	Integration into nightly build		11/27/08	12/01/08	4
2.2.2.5	Check final results for accuracy		12/01/08	12/04/08	3
2.0.0.0	**Line Scanner**	R. Searle	12/04/08	2/13/09	71
2.1.0.0	**Single Image Save**		12/04/08	1/10/09	18
2.1.1.0	**Geocorrected Imagery**		12/04/08	12/24/08	18
2.1.1.1	Research & Understanding Code		12/04/08	12/07/08	3
2.1.1.2	Write unit test code		12/10/08	12/13/08	3
2.1.1.3	Accuracy Testing		12/14/08	12/17/08	3
2.1.1.4	Integration into nightly build		12/17/08	12/21/08	4
2.1.1.5	Check final results for accuracy		12/21/08	12/24/08	3
2.0.2.0	**Non-Geocorrected**				
2.0.2.1	Research & Understanding Code		12/24/08	12/27/08	3
2.0.2.2	Write unit test code		12/27/08	12/31/08	4
2.0.2.3	Accuracy Testing		12/31/08	1/03/09	3
2.0.2.4	Integration into nightly build		1/03/09	1/07/09	4
2.0.2.5	Check final results for accuracy		1/07/09	1/10/09	3
2.2.0.0	**Multi-Image save**				
2.2.1.0	**Geocorrected Imagery**				
2.2.1.1	Research & Understanding Code		1/10/09	1/13/09	3
2.2.1.2	Write unit test code		1/13/09	1/17/09	4
2.2.1.3	Accuracy Testing		1/17/09	1/20/09	3
2.2.1.4	Integration into nightly build		1/20/09	1/24/09	4
2.2.1.5	Check final results for accuracy		1/24/09	1/27/09	3
2.2.2.0	**Non-Geocorrected**				
2.2.2.1	Research & Understanding Code		1/27/09	1/30/09	3
2.2.2.2	Write unit test code		1/30/09	2/03/09	4
2.2.2.3	Accuracy Testing		2/03/09	2/06/09	3
2.2.2.4	Integration into nightly build		2/06/09	2/10/09	4
2.2.2.5	Check final results for accuracy		2/10/09	2/13/09	3

Figure 13: Gantt Chart 2



WBS	Tasks	Task Lead	Start	End	Duration (Days)
3.1.1.0	**Color Framing**	A. Crookston	9/22/08	12/04/08	73
3.1.1.1	**Single Image Save**		9/22/08	10/31/08	39
3.1.1.2	**Geocorrected Imagery**		9/22/08	10/14/08	22
3.1.1.3	Research & Understanding Code		9/22/08	9/26/08	4
3.1.1.4	Write unit test code		9/29/08	10/03/08	4
3.1.1.5	Accuracy Testing		10/04/08	10/07/08	3
3.1.1.6	Integration into nightly build		10/07/08	10/11/08	4
3.1.1.7	Check final results for accuracy		10/11/08	10/14/08	3
3.1.2.0	**Non-Geocorrected**				
3.1.2.1	Research & Understanding Code		10/14/08	10/17/08	3
3.1.2.2	Write unit test code		10/17/08	10/21/08	4
3.1.2.3	Accuracy Testing		10/21/08	10/24/08	3
3.1.2.4	Integration into nightly build		10/24/08	10/28/08	4
3.1.2.5	Check final results for accuracy		10/28/08	10/31/08	3
3.2.0.0	**Multi-Image save**				
3.2.1.0	**Geocorrected Imagery**				
3.2.1.1	Research & Understanding Code		10/31/08	11/03/08	3
3.2.1.2	Write unit test code		11/03/08	11/07/08	4
3.2.1.3	Accuracy Testing		11/07/08	11/10/08	3
3.2.1.4	Integration into nightly build		11/10/08	11/14/08	4
3.2.1.5	Check final results for accuracy		11/14/08	11/17/08	3
3.2.2.0	**Non-Geocorrected**				
3.2.2.1	Research & Understanding Code		11/17/08	11/20/08	3
3.2.2.2	Write unit test code		11/20/08	11/24/08	4
3.2.2.3	Accuracy Testing		11/24/08	11/27/08	3
3.2.2.4	Integration into nightly build		11/27/08	12/01/08	4
3.2.2.5	Check final results for accuracy		12/01/08	12/04/08	3
3.1.1.0	**SAR**	A. Crookston	12/04/08	2/27/09	#REF!
3.1.1.1	**Single Image Save**		12/04/08	1/24/09	51
3.1.1.2	**Geocorrected Imagery**		12/04/08	1/07/09	34
3.1.1.3	Research & Understanding Code		12/04/08	12/07/08	3
3.1.1.4	Write unit test code		12/10/08	12/14/08	4
3.1.1.5	Accuracy Testing		12/15/08	12/18/08	3
3.1.1.6	Integration into nightly build		12/18/08	12/22/08	4
3.1.1.7	Check final results for accuracy		1/04/09	1/07/09	3
3.1.2.0	**Non-Geocorrected**				
3.1.2.1	Research & Understanding Code		1/07/09	1/10/09	3
3.1.2.2	Write unit test code		1/10/09	1/14/09	4
3.1.2.3	Accuracy Testing		1/14/09	1/17/09	3
3.1.2.4	Integration into nightly build		1/17/09	1/21/09	4
3.1.2.5	Check final results for accuracy		1/21/09	1/24/09	3
3.2.0.0	**Multi-Image save**				
3.2.1.0	**Geocorrected Imagery**				
3.2.1.1	Research & Understanding Code		1/24/09	1/27/09	3
3.2.1.2	Write unit test code		1/27/09	1/31/09	4
3.2.1.3	Accuracy Testing		1/31/09	2/03/09	3
3.2.1.4	Integration into nightly build		2/03/09	2/07/09	4
3.2.1.5	Check final results for accuracy		2/07/09	2/10/09	3
3.2.2.0	**Non-Geocorrected**				
3.2.2.1	Research & Understanding Code		2/10/09	2/13/09	3
3.2.2.2	Write unit test code		2/13/09	2/17/09	4
3.2.2.3	Accuracy Testing		2/17/09	2/20/09	3
3.2.2.4	Integration into nightly build		2/20/09	2/24/09	4
3.2.2.5	Check final results for accuracy		2/24/09	2/27/09	3

Figure 14: Gantt Chart 3

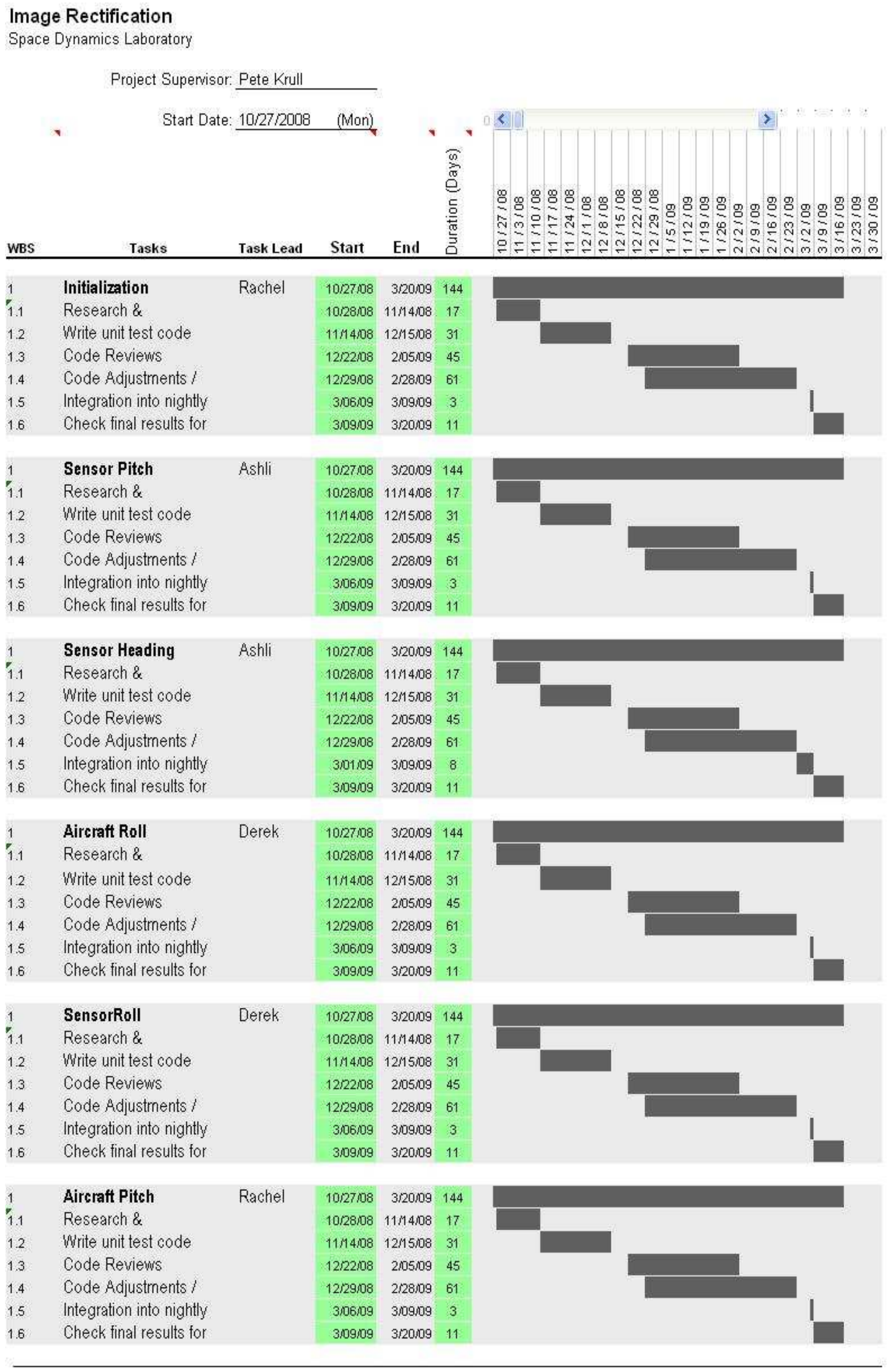


Image Rectification
Space Dynamics Laboratory

Project Supervisor: Pete Krull

Start Date: 10/27/2008 (Mon)

WBS	Tasks	Task Lead	Start	End	Duration (Days)
1	**Initialization**	Rachel	10/27/08	3/20/09	144
1.1	Research &		10/28/08	11/14/08	17
1.2	Write unit test code		11/14/08	12/15/08	31
1.3	Code Reviews		12/22/08	2/05/09	45
1.4	Code Adjustments /		12/29/08	2/28/09	61
1.5	Integration into nightly		3/06/09	3/09/09	3
1.6	Check final results for		3/09/09	3/20/09	11
1	**Sensor Pitch**	Ashli	10/27/08	3/20/09	144
1.1	Research &		10/28/08	11/14/08	17
1.2	Write unit test code		11/14/08	12/15/08	31
1.3	Code Reviews		12/22/08	2/05/09	45
1.4	Code Adjustments /		12/29/08	2/28/09	61
1.5	Integration into nightly		3/06/09	3/09/09	3
1.6	Check final results for		3/09/09	3/20/09	11
1	**Sensor Heading**	Ashli	10/27/08	3/20/09	144
1.1	Research &		10/28/08	11/14/08	17
1.2	Write unit test code		11/14/08	12/15/08	31
1.3	Code Reviews		12/22/08	2/05/09	45
1.4	Code Adjustments /		12/29/08	2/28/09	61
1.5	Integration into nightly		3/01/09	3/09/09	8
1.6	Check final results for		3/09/09	3/20/09	11
1	**Aircraft Roll**	Derek	10/27/08	3/20/09	144
1.1	Research &		10/28/08	11/14/08	17
1.2	Write unit test code		11/14/08	12/15/08	31
1.3	Code Reviews		12/22/08	2/05/09	45
1.4	Code Adjustments /		12/29/08	2/28/09	61
1.5	Integration into nightly		3/06/09	3/09/09	3
1.6	Check final results for		3/09/09	3/20/09	11
1	**SensorRoll**	Derek	10/27/08	3/20/09	144
1.1	Research &		10/28/08	11/14/08	17
1.2	Write unit test code		11/14/08	12/15/08	31
1.3	Code Reviews		12/22/08	2/05/09	45
1.4	Code Adjustments /		12/29/08	2/28/09	61
1.5	Integration into nightly		3/06/09	3/09/09	3
1.6	Check final results for		3/09/09	3/20/09	11
1	**Aircraft Pitch**	Rachel	10/27/08	3/20/09	144
1.1	Research &		10/28/08	11/14/08	17
1.2	Write unit test code		11/14/08	12/15/08	31
1.3	Code Reviews		12/22/08	2/05/09	45
1.4	Code Adjustments /		12/29/08	2/28/09	61
1.5	Integration into nightly		3/06/09	3/09/09	3
1.6	Check final results for		3/09/09	3/20/09	11

Figure 15: Gantt Chart 4

9.4 Facilities

Our project was completed using the Space Dynamics Laboratory's North Logan facilities. "SDL is headquartered in the 140,000-square foot Jake Garn Space Research Complex near the Utah State University campus in Logan, Utah." They are located at:

1695 North Research Park Way
North Logan, Utah
84341

Figure 16: Space Dynamics Laboratory Facilities in North Logan

9.5 Personnel

The personnel used to complete this project's design were Ashli Crookston and Derek Hampton, who are seniors in Computer Engineering at Utah State University, and Rachel Searle, who is a senior in Electrical Engineering at Utah State University. All three work as software testers at the Space Dynamics Laboratory.

9.6 Work Breakdown Structure

The following diagram shows the work breakdown structure for our project. The project can be broken into five phases: design, research, coding, review, and implementation.

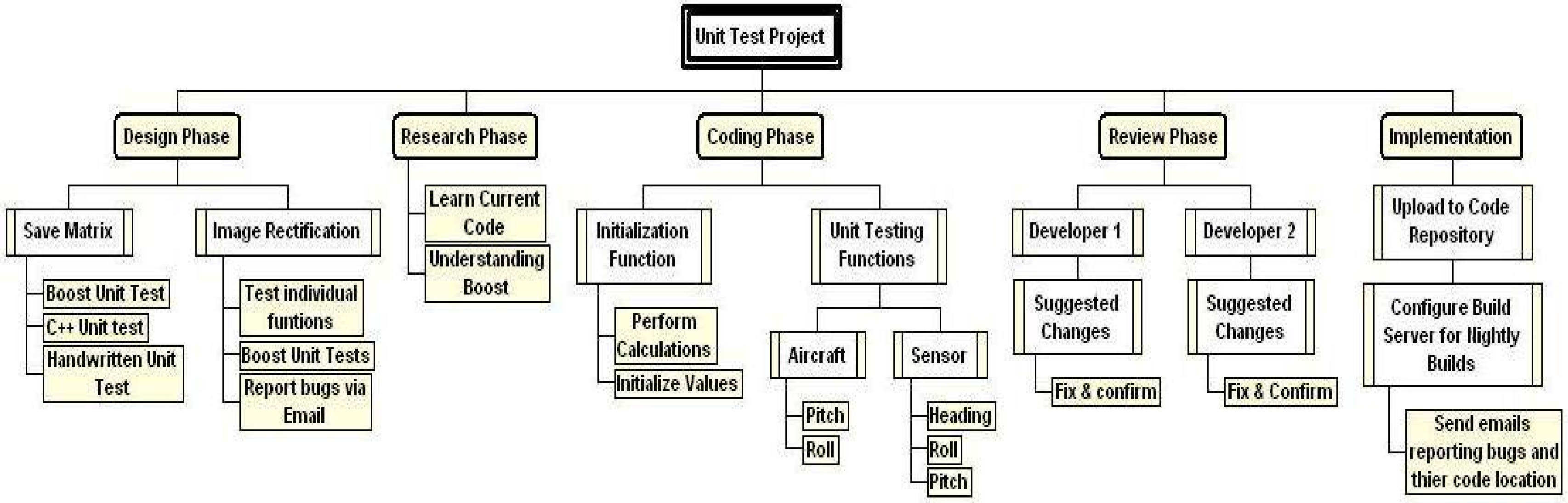


Figure 17: Work Breakdown Structure

Within the design phase, we looked at the Save Matrix as a test subject and researched Boost C++Test, C++Test, and handwritten unit tests. In the design phase, we also switched to the image rectification project and looked into different ways of implementing it. During the research phase of our project, we familiarized ourselves with the existing code and worked on understanding how to use the Boost C++ Test Libraries. During the coding phase, we wrote the initialization function, which performs calculations and initializes values needed throughout the code. We also wrote our unit testing functions, which included aircraft pitch and roll as well as sensor heading, roll, and pitch. In the review phase, we went through code reviews with developers one and two, whom suggested changes to our code, let us fix it, and then approved it. In the implementation phase, we uploaded our code to the repository, configured the build server to run our code in the nightly build, and made sure that error emails were sent out to the developers.

10.0 Conclusion

10.1 Purpose of Report

The purpose of this report was to discuss the background and need for our project, a detailed description of the design and implementation of our project and the results, and also the work breakdown and cost estimates for our project.

10.2 Objectives

The Space Dynamics Laboratory hires testers to manually test each function of their software. This is time consuming and inefficient. Our objective was to create a better method of testing by implementing an automated testing system. This involved writing unit test code created with the Boost Test Libraries, integrating this code into the nightly build server, and allowing the system to email the error reports generated by the nightly build to code owners so that they might be able to find and fix their errors as quickly as possible.

10.3 Summary of Final Design Selection

We were able to successfully complete our project. Our unit testing code compiles and works correctly. The code checks for potential problems created by changes to the image rectification code. Emails are sent out to the appropriate developer if an error is introduced. Our code is very important because it helps to ensure that a key functionality of the C4ISR division's software is working error-free before it is sent to the customer. Our managers and the software developers are very pleased with our work.

10.4 Costs and Timeline

Since all of the software and systems used for our project were freeware or already paid for or owned by SDL, we did not have to make any purchases throughout the course of our project. The only project-related charges to SDL were the man hours that we put into the project. Since we worked on our project during normal working hours, SDL didn't have to pay us any extra on top of our normal hourly wages for this project. When the time charged for this project was added up, it totaled to about $10,000.

Figure 18 shows a timeline of our projects milestones. It covers events occurring between April, 2008, and March, 2009. The timeline is broken up into five main phases: design, research, coding, review, and implementation.

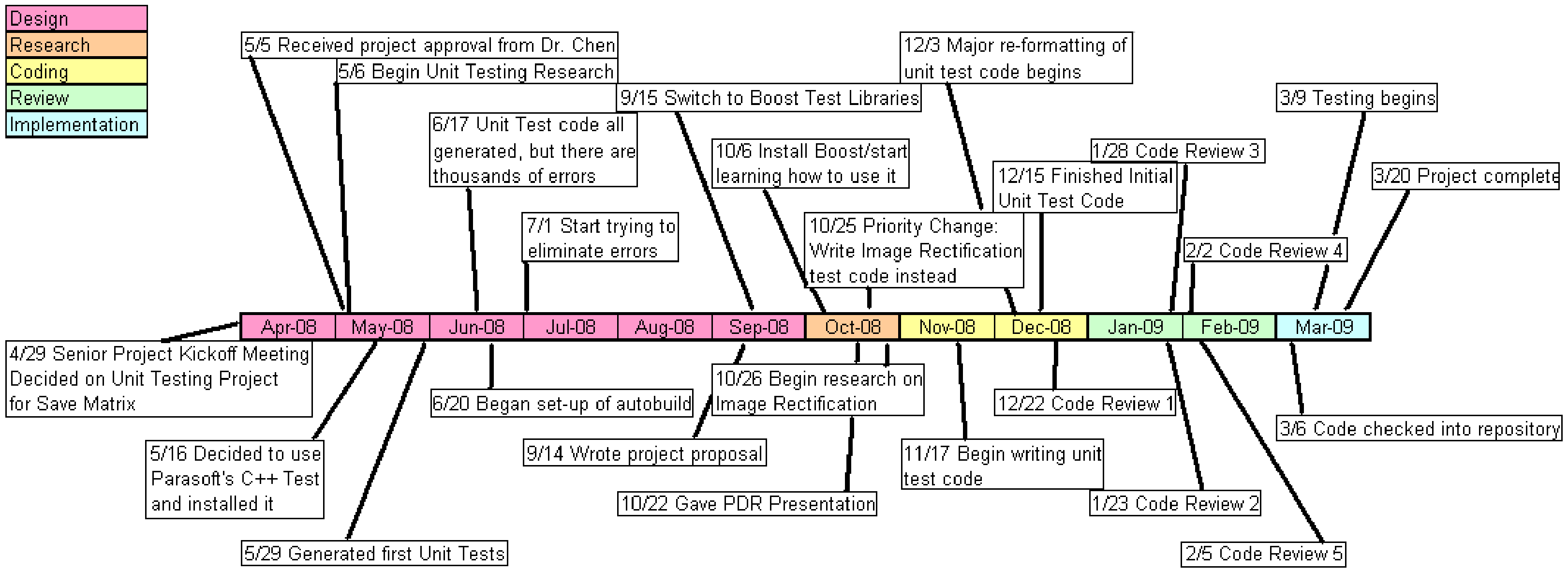

Design
Research
Coding
Review
Implementation
Apr-08
May-08
Jun-08
Jul-08
Aug-08
Sep-08
Oct-08
Nov-08
Dec-08
Jan-09
Feb-09
Mar-09
4/29 Senior Project Kickoff Meeting Decided on Unit Testing Project for Save Matrix
5/5 Received project approval from Dr. Chen
5/6 Begin Unit Testing Research
5/16 Decided to use Parasoft's C++ Test and installed it
5/29 Generated first Unit Tests
6/17 Unit Test code all generated, but there are thousands of errors
6/20 Began set-up of autobuild
7/1 Start trying to eliminate errors
9/14 Wrote project proposal
9/15 Switch to Boost Test Libraries
10/6 Install Boost/start learning how to use it
10/22 Gave PDR Presentation
10/25 Priority Change: Write Image Rectification test code instead
10/26 Begin research on Image Rectification
11/17 Begin writing unit test code
12/3 Major re-formatting of unit test code begins
12/15 Finished Initial Unit Test Code
12/22 Code Review 1
1/23 Code Review 2
1/28 Code Review 3
2/2 Code Review 4
2/5 Code Review 5
3/6 Code checked into repository
3/9 Testing begins
3/20 Project complete

Figure 18: Project Timeline

Appendices:

A. Bibliography

1. "C4ISR Systems." Products and Capabilities.2009. The Space Dynamics Laboratory. <http://www.sdl.usu.edu/products-capabilities/c4isr>. <http://www.sdl.usu.edu/products-capabilities/c4isr>.
2. "The Electromagnetic Spectrum." Glossary of Terms. Laboratory for Computational Science and Engineering. <www.lcse.umn.edu>.
3. "Vantage Software Suite." Products and Capabilities. 2009 The Space Dynamics Laboratory. <http://www.sdl.usu.edu/products-capabilities/vantage>.
4. DeChristopher, Robert. "Roll, Pitch, and Yaw." Mr. D.'s World of Math and Science. November 14, 2005. <http://fifthpostulate.net/roll_pitch_and_yaw.htm>.
5. "Logan, UT Facility." About SDL.2009. The Space Dynamics Laboratory. <http://www.sdl.usu.edu/about/logan-facility>.
6. "C++Test Product Overview". Parasoft C++Test. Parasoft. 2009. <http://www.parasoft.com/jsp/products/home.jsp?product=Wizard&/>.
7. Teo, Y.M., Tay, S.C., and Gozali, J.P. "Distributed Geo-rectification of Satellite Images using Grid Computing". Centre for Remote Imaging, Sensing and Processing. Department of Computer Science, National University of Singapore. April 2003. <http://www.comp.nus.edu.sg/~teoym/pub/03/ipdps03.pdf>.

B. Supporting Documents for Project Management

i. Gant Chart

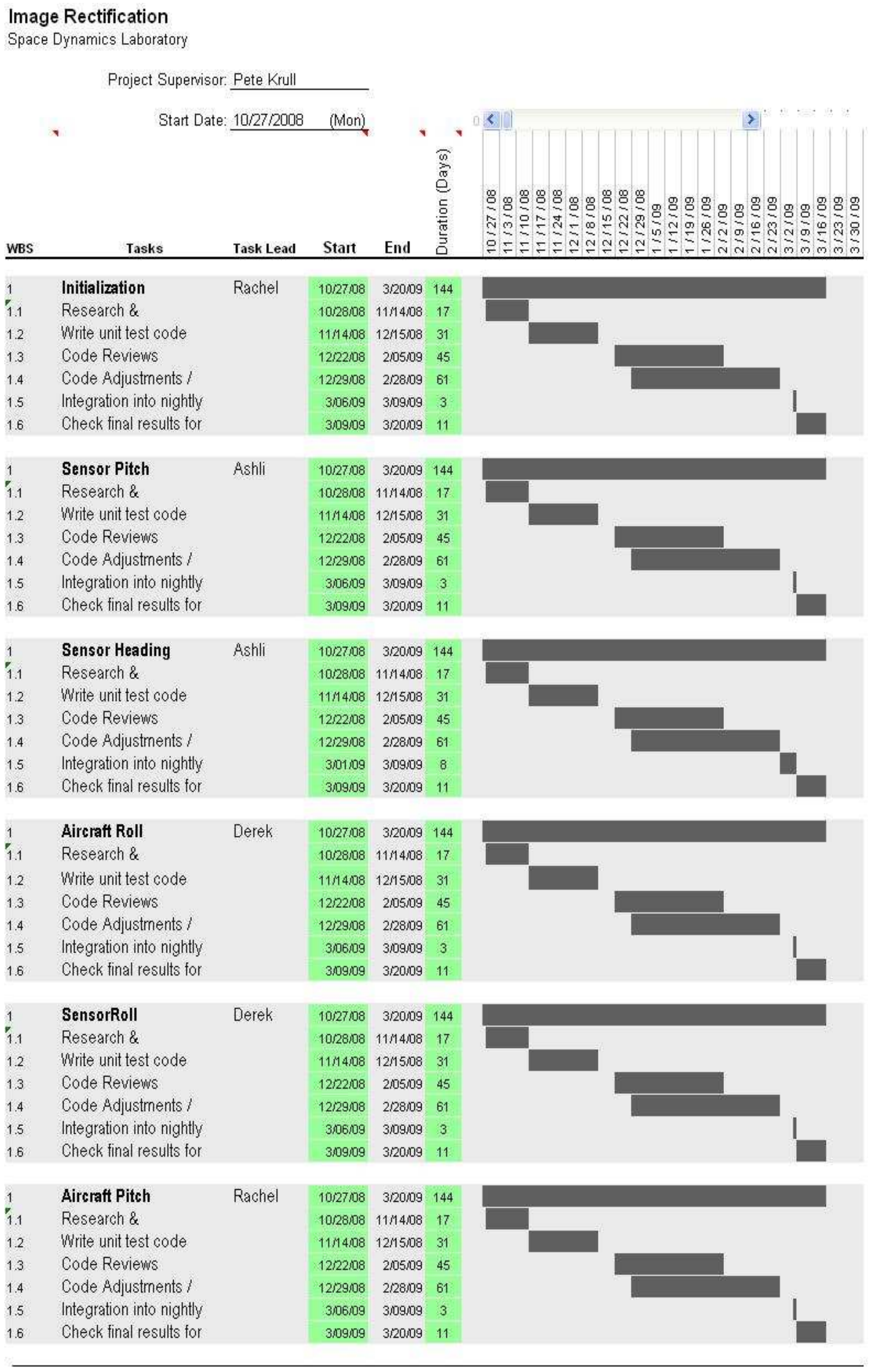


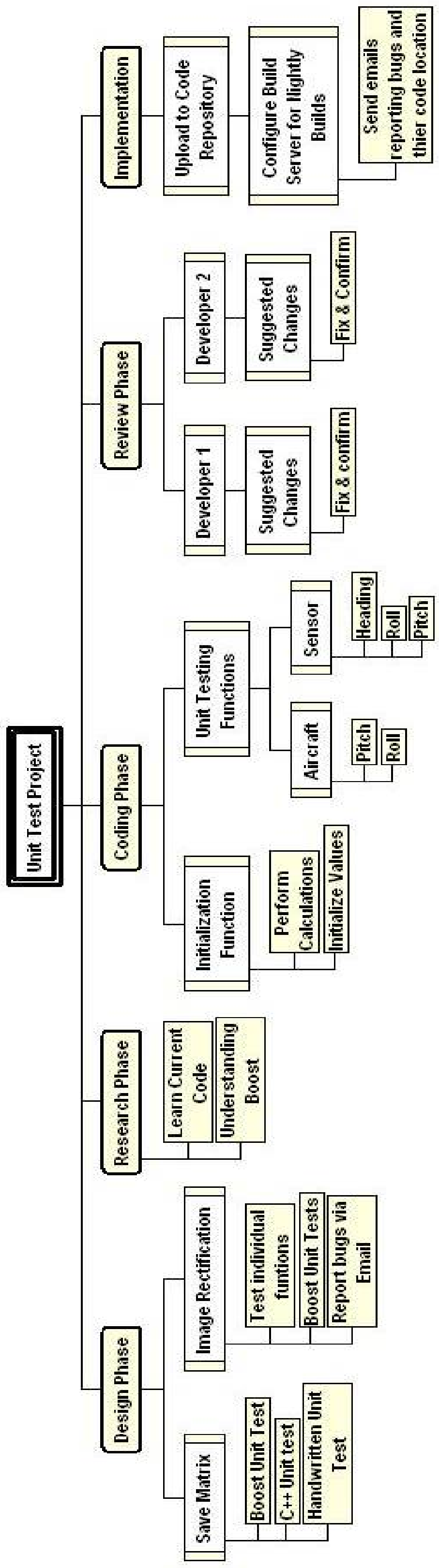
Image Rectification
Space Dynamics Laboratory

Project Supervisor: Pete Krull

Start Date: 10/27/2008 (Mon)

WBS	Tasks	Task Lead	Start	End	Duration (Days)
1	**Initialization**	Rachel	10/27/08	3/20/09	144
1.1	Research &		10/28/08	11/14/08	17
1.2	Write unit test code		11/14/08	12/15/08	31
1.3	Code Reviews		12/22/08	2/05/09	45
1.4	Code Adjustments /		12/29/08	2/28/09	61
1.5	Integration into nightly		3/06/09	3/09/09	3
1.6	Check final results for		3/09/09	3/20/09	11
1	**Sensor Pitch**	Ashli	10/27/08	3/20/09	144
1.1	Research &		10/28/08	11/14/08	17
1.2	Write unit test code		11/14/08	12/15/08	31
1.3	Code Reviews		12/22/08	2/05/09	45
1.4	Code Adjustments /		12/29/08	2/28/09	61
1.5	Integration into nightly		3/06/09	3/09/09	3
1.6	Check final results for		3/09/09	3/20/09	11
1	**Sensor Heading**	Ashli	10/27/08	3/20/09	144
1.1	Research &		10/28/08	11/14/08	17
1.2	Write unit test code		11/14/08	12/15/08	31
1.3	Code Reviews		12/22/08	2/05/09	45
1.4	Code Adjustments /		12/29/08	2/28/09	61
1.5	Integration into nightly		3/01/09	3/09/09	8
1.6	Check final results for		3/09/09	3/20/09	11
1	**Aircraft Roll**	Derek	10/27/08	3/20/09	144
1.1	Research &		10/28/08	11/14/08	17
1.2	Write unit test code		11/14/08	12/15/08	31
1.3	Code Reviews		12/22/08	2/05/09	45
1.4	Code Adjustments /		12/29/08	2/28/09	61
1.5	Integration into nightly		3/06/09	3/09/09	3
1.6	Check final results for		3/09/09	3/20/09	11
1	**SensorRoll**	Derek	10/27/08	3/20/09	144
1.1	Research &		10/28/08	11/14/08	17
1.2	Write unit test code		11/14/08	12/15/08	31
1.3	Code Reviews		12/22/08	2/05/09	45
1.4	Code Adjustments /		12/29/08	2/28/09	61
1.5	Integration into nightly		3/06/09	3/09/09	3
1.6	Check final results for		3/09/09	3/20/09	11
1	**Aircraft Pitch**	Rachel	10/27/08	3/20/09	144
1.1	Research &		10/28/08	11/14/08	17
1.2	Write unit test code		11/14/08	12/15/08	31
1.3	Code Reviews		12/22/08	2/05/09	45
1.4	Code Adjustments /		12/29/08	2/28/09	61
1.5	Integration into nightly		3/06/09	3/09/09	3
1.6	Check final results for		3/09/09	3/20/09	11

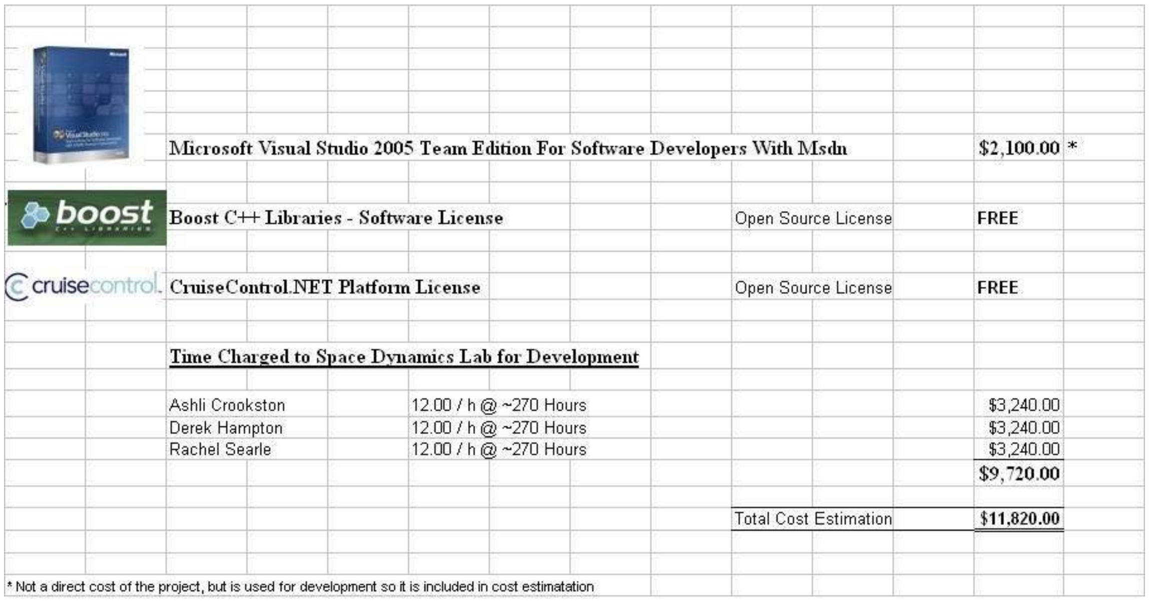
ii. WBS


Unit Test Project
Design Phase
Save Matrix
Boost Unit Test
C++ Unit test
Handwritten Unit Test
Image Rectification
Test individual funtions
Boost Unit Tests
Report bugs via Email
Research Phase
Learn Current Code
Understanding Boost
Coding Phase
Initialization Function
Perform Calculations
Initialize Values
Unit Testing Functions
Aircraft
Pitch
Roll
Sensor
Heading
Roll
Pitch
Review Phase
Developer 1
Suggested Changes
Fix & confirm
Developer 2
Suggested Changes
Fix & Confirm
Implementation
Upload to Code Repository
Configure Build Server for Nightly Builds
Send emails reporting bugs and thier code location

iii. *Engineering Design Task List*

*Write initialization function-Rachel
*Write unit testing functions
 -Sensor pitch function-Ashli
 -Sensor roll function -Derek
 -Sensor heading function -Ashli
 -Aircraft pitch function -Rachel
 -Aircraft roll function-Derek

iv. *Cost Breakdown info*



Microsoft Visual Studio 2005 Team Edition For Software Developers With Msdn			**$2,100.00**	*
Boost C++ Libraries - Software License		Open Source License	**FREE**	
CruiseControl.NET Platform License		Open Source License	**FREE**	
<u>Time Charged to Space Dynamics Lab for Development</u>				
Ashli Crookston	12.00 / h @ ~270 Hours		$3,240.00	
Derek Hampton	12.00 / h @ ~270 Hours		$3,240.00	
Rachel Searle	12.00 / h @ ~270 Hours		$3,240.00	
			$9,720.00	
		Total Cost Estimation	**$11,820.00**	

* Not a direct cost of the project, but is used for development so it is included in cost estimatation