3-2005

A Three Dimensional Helmet Mounted Primary Flight Reference for Paratroopers

Jason I. Thompson

A THREE DIMENSIONAL HELMET MOUNTED PRIMARY FLIGHT REFERENCE FOR PARATROOPERS

THESIS

Jason I. Thompson, Second Lieutenant, USAF

AFIT/GCS/ENG/05-18

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

AFIT/GCS/ENG/05-18

A THREE DIMENSIONAL HELMET MOUNTED PRIMARY FLIGHT REFERENCE FOR PARATROOPERS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Jason I. Thompson, BS

Second Lieutenant, USAF

March 2005

AFIT/GCS/ENG/05-18

A THREE DIMENSIONAL HELMET MOUNTED PRIMARY FLIGHT REFERENCE FOR PARATROOPERS

Jason I. Thompson, BS

Second Lieutenant, USAF

Approved:

/signed/

_____ _____

Dr. John F. Raquet (Chairman) Date

/signed/

_____ _____

Major Robert P. Graham (Member) Date

/signed/

_____ _____

Major Joel D. Young (Member) Date

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Dr. John Raquet, for his guidance and support throughout the course of this thesis effort. His encouragement, insight, and enthusiasm made this thesis possible. I would also like to thank my sponsor, Rob Stephens, from Specialized Technical Services, for both the support and latitude provided to me in this endeavor. Letting me use, hands on, some of the neat gear they make was a definite perk.

Thank you as well to Don and Jay for building the computer hardware and GPS software that this thesis ran on. Both of you gave me invaluable advice that helped immensely.

Thanks to those who helped me test the software. I know walking around in the cold, squishing through the mud, and huddling against the wind, all while squinting at a little HUD, wasn't terribly fun, but without you, I wouldn't have finished this thesis.

I would also like to thank my fantastic fiancé and soon to be wife. Her support was invaluable. I couldn't have done this without her.

Jason I. Thompson

Table of Contents

Page

Acknowledgments iv
Table of Contents v
List of Figures ix
List of Tables xii
Abstract xiii
1. Introduction 1
1.1. Introduction 1
1.2. Background 2
1.3. Research Focus 5
1.4. Objectives 6
1.5. Assumptions 7
1.6. Approach 7
1.7. Document Overview 8
2. Literature Review 9
2.1. Introduction 9
2.2. Navigation with GPS 9
2.2.1. Related Research in GPS Navigation 15
2.3. Helmet Mounted Displays 19
2.3.1. Related Research in Helmet Mounted Displays 22
2.4. Parachute Navigation 25
2.4.1. Related Research in Parachute Navigation 27
2.5. Tunnel in The Sky Navigation Displays 32

2.5.1. Related Research in Tunnel in the Sky Navigation Displays 34

2.6. Summary 37

3. System Description 39

3.1. Introduction 39

3.2. Coordinate Systems 39

3.3. Software Design 40

3.3.1. Description of Allegro 40

3.3.2. Description of the Information Supplying Software 42

3.4. Sequence of Actions 43

3.5. Path Generating Algorithm 47

3.5.1. Description of the Path Generating Algorithm 47

3.5.2. Analysis of the Performance of the Path Generating Algorithm 52

3.5.3. Weaknesses of the Path Generating Algorithm 54

3.6. Generating the Virtual World 56

3.6.1. Waypoints 57

3.6.2. The Path 57

3.6.3. Weaknesses of the Virtual World's Implementation 58

3.7. Design of the HUD 59

3.7.1. Elements of the HUD 60

3.8. Summary 62

4. Methodology 63

4.1. Chapter Overview 63

4.1.1. Goals and Hypothesis 63

4.1.2. Approach 65
4.2. System Description 65
4.3. Evaluation Technique 66
4.4. Experimental Design 69
4.5. System Services 71
4.6. Workload 71
4.7. Performance Metrics 72
4.8. Test Parameters 72
4.8.1. Test Factors 72
4.9. Suggestions for Future Testing 73
4.10. Summary 75
5. Test Results 77
5.1. Introduction 77
5.2. Test Design 77
5.3. Path Configuration 78
5.4. Test Results 79
5.5. Analysis 82
5.6. Conclusion 84
6. Conclusions and Recommendations 86
6.1. Conclusions of Research 86
6.2. Significance of Research 87
6.3. Recommendations for Future Research 88
Appendix A – HUD Source Code Used for Testing 91

Appendix B – Effects of the AN/PVS-21 NVGs On the MT-9 IMU 119
Bibliography 126

List of Figures

Page

Figure 1: Current Text-Based HUD Displayed On NVGs [STS03b]..... 4

Figure 2: GPS Ground Control Segment Locations [ACK94] 10

Figure 3: The Intersection of Three Circles Defines a Single Point 13

Figure 4: Speed Error For A Bicycle Traveling In A Straight Line [WIT04]. 16

Figure 5: GPS Reported Speed Vs. Actual Speed For A Straight Line [WIT04]. Black Circles Represent GPS Speed; White Circles Represent Actual Speed. 17

Figure 6: Speed Error For A Bicycle Traveling In A Large Circular Path [WIT04]..... 17

Figure 7: Speed Error For A Bicycle Traveling In A Small Circular Path [WIT04]..... 18

Figure 8: GPS Speed Vs Actual Speed For A Curved Path [WIT04]. 19

Figure 9: Actual and Estimated Errors in Position of the IMU [JOF03]. 23

Figure 10: Actual and Estimated Error in Velocity of the IMU [JOF03]. 24

Figure 11: Holding Crab [GER04]. 27

Figure 12: Photograph Of The OPANAS System In Flight [SSK05]. 28

Figure 13: Early, Late, And Middle Potential Energy Management Paths For The Pegasus GNC Algorithm [KAM03]..... 31

Figure 14: MIL-STD-1787 HUD [NEW00]. 32

Figure 15: Different Tunnel in the Sky Configurations Tested By [PRI03]..... 36

Figure 16: The Three Worlds That Play A Role In The HUD System..... 39

Figure 17: Sequence Of Events During Program Execution 44

Figure 18: ‘Close’ And ‘Far Away’ Areas (Areas Shaded Green Are ‘Close’). 46

Figure 19: Example Inputs And Outputs To The Path Generating Algorithm 48

Figure 20: Pseudocode For The PGA 49

Figure 21: Top-Down View Of The PGA's Progression 51

Figure 22: Side View Of The PGA's Progression..... 52

Figure 23: Simplified Pseudocode of the PGA..... 53

Figure 24: Simulated View Of The HUD Through The NVGs..... 60

Figure 25: Simulated HUD Displays 60

Figure 26: View Through The NVGs Without Any Navigational Aids. The User Will Be Equipped With A GPS Displaying The Latitude, Longitude, And Direction Of Travel. 63

Figure 27: View Through The NVGs Using A Tunnel In The Sky System. By Flying Through The Tunnel, Which Depicts The Generated Path To LZ, The Paratrooper Will Be Guided To The LZ. If The Tunnel Is Not Visible, An Arrow Will Appear At The Side Of The Screen. 63

Figure 28: The System Under Test 66

Figure 29: Path Layouts Used For Testing. Green Circles Indicate Start Points; Red Circles Indicate Stop Points. 79

Figure 30: User Trajectories For Path 1. Green Lines Represent GPS-Based Paths And Red Lines Represent HUD-Based Paths. The Blue-And-Gray Dashed Line Indicates The Straight Line Path Between Waypoints. The Green Circle Represents The Starting Waypoint And The Red Circle Represents The Ending Waypoint. 80

Figure 31: User trajectories for Path 2. Green Lines Represent GPS-Based Paths And Blue Lines Represent HUD-Based Paths. The Blue-And-Gray Dashed Line Indicates The Straight Line Path Between Waypoints. The Green Circle Represents The Starting Waypoint And The Red Circle Represents The Ending Waypoint 81
Figure 32: Users With The HUD (Red Lines) Tended To Cut Corners At Turns In The Path (Blue And Gray Line). 81
Figure 33: Track Of A User With HUD Losing Sight Of The Path. The Black Arrows Indicate The Direction Of Travel. The Green Line Represents The Trajectory Of A User With GPS. The Red Line Represents The Trajectory Of A User With The HUD. The Blue And Gray Line Represents The Path. 83
Figure 34: 90% Confidence Intervals for the Various NVG States 122
Figure 35: Average Sensor Readings For Various NVG States 123
Figure 36: Gyroscope Measurements During The Baseline Experiment. 124
Figure 37: Gyroscope Measurements While The NVGS Were Turned On, But Covered. 124

List of Tables

Page

Table 1: Time Taken By Each User To Walk Each Path 79

Table 2: Average Times For A User To Navigate A Path. 82

Table 3: Average values recorded for each sensor 120

Table 4: ANOVA conducted on the data obtained during the experiments 121

Table 5: Mean Effect, standard deviation, and confidence intervals for various states of the NVGs 121

Abstract

This thesis seeks to develop a Heads Up Display (HUD) presented on a Helmet Mounted Display (HMD), which presents a three-dimensional, graphical, predictive navigational reference to a paratrooper during a High Altitude, High Opening (HAHO) parachute jump. A Path Generating Algorithm (PGA) takes as input the Landing Zone's (LZ) location, the wind profile, and the paratrooper's parachute's performance characteristics, and returns a set of waypoints for the paratrooper to follow. The PGA attempts to maximize the distance that the paratrooper travels.

The PGA's output is used to build a path to the LZ from a Release Point (RP). During the jump, GPS signals and an Inertial Measurement Unit functioning as a head tracker is used to determine the user's location and head orientation. The HUD presents a virtual "Tunnel in the Sky" for the paratrooper to navigate through.

Due to hardware unavailability, the head tracker could not be implemented. Ground testing of the system sans the head tracker determined that subjects using the graphical HUD navigated a path slower than when compared to a baseline navigational reference. It is theorized that implementing the head tracker will make the HUD more efficient and better suited to navigation.

A THREE DIMENSIONAL HELMET MOUNTED PRIMARY FLIGHT REFERENCE FOR PARATROOPERS

1. Introduction

1.1. Introduction

Ever since the parachute was introduced as a viable tool for military forces, they have been used to accomplish goals once thought unattainable. Parachutes can save a pilot bailing out from a burning aircraft, or be used to deliver troops deep behind enemy lines. Parachutes can airdrop vital supplies, both to civilians during a humanitarian mission, or to beleaguered troops surrounded by the enemy. Advances in parachute canopy technology and navigation techniques constantly add to the capabilities of the parachute.

For the United States Air Force, one of the most important parachute missions conducted involves pararescuemen, or PJs. These specially trained airmen specialize in covertly entering territory in order to rescue pilots who have been shot down. Although they can employ many entry tactics, High Altitude, High Opening (HAHO) parachute jumps are a preferred method. HAHO jumps are conducted at altitudes up to 30,000 feet, with the parachute opening occurring shortly after the jump. With the right parachute and winds, a PJ can travel long distances to a landing zone [STS03a]. This standoff range is often combined with the cover of night so the PJs can evade enemy detection.

The PJs' mission is important, both for the preservation of military capability and for the morale boost that it provides to pilots. Bringing a military pilot to full mission capability is an expensive and time-consuming process. Rescuing a pilot, rather than

abandoning him, allows the Air Force to bring him back into the fight in days, rather than waiting years for a replacement pilot to arrive. Beyond this, though, lies the fact that there isn't an airman who would willingly leave a fellow airman behind enemy lines. PJs perform a vital, but hazardous, task.

1.2. Background

In order for a PJ to come to the aid of a downed airman, they must first reach the airman, and methods to do so have evolved over time. The first HAHO jump was conducted by USAF Colonel Joe Kittinger at an altitude of 19 miles above the earth's surface. The first use of a HAHO jump in a combat operation occurred during the Vietnam War. While HAHO jumps solved the problem of getting to a stranded airman, other problems arose. For instance, conducting a HAHO jump during daylight meant that an enemy on the ground could easily observe the paratroopers and where they landed.

To avoid enemy detection, HAHO jumps at night are utilized, but this introduces new problems. During a HAHO jump, paratroopers navigate using a compass and by visual confirmation of landmarks. Navigation by landmarks is difficult or impossible at night. In addition, landing without being able to clearly see the ground could spell disaster. These problems can be overcome by the use of Night Vision Goggle (NVG) technology. NVGs amplify available light, such as moonlight, starlight, and ambient light from cities, enabling a person to see much more clearly during times of darkness. By applying this technology, paratroopers can avoid detection by the enemy, and utilize the element of surprise when conducting their missions.

NVGs are not perfect. They only amplify available light and cannot see through obscurants, such as clouds, smoke, and fog. During a very dark night, such as one with no moonlight, landmarks can be impossible to see from high altitudes, making navigation impossible. If clouds are present, the paratrooper cannot see the ground when he is above the cloud layer, again making navigation by landmarks impossible. At lower altitudes, when the paratrooper is inside of the cloud cover, disorientation can occur due to the complete lack of visual cues. Because of these problems, paratroopers must abort a mission if excessive cloud cover is present.

Being able to conduct missions despite inclement weather would greatly improve the capabilities of paratroopers. Thick clouds can provide excellent concealment from the eyes of any enemy troops on the ground. Also, since it is currently impossible to conduct parachute operations during bad weather, paratroopers could insert themselves when and where the enemy is not expecting them. Perhaps the most important reason for being able to overcome the weather is the fact that missions will no longer have to be cancelled because of cloud cover. While all military operations are important, the PJs' mission of rescuing downed pilots is a particularly time sensitive one. If the PJs cannot make a timely HAHO jump, the pilot may be captured or killed. Therefore, it is very important that some method of being able to navigate through cloud cover is developed.

Current methods of presenting navigational information to paratroopers combine Global Positioning System (GPS) technology with Heads Up Display (HUD) technology. By displaying precise navigational information to a paratrooper continuously on a HUD, an attempt was made to overcome cloud cover. Unfortunately, current display formats

are hard to understand due to their text-based nature. Figure 1 illustrates a current text-based HUD layout.

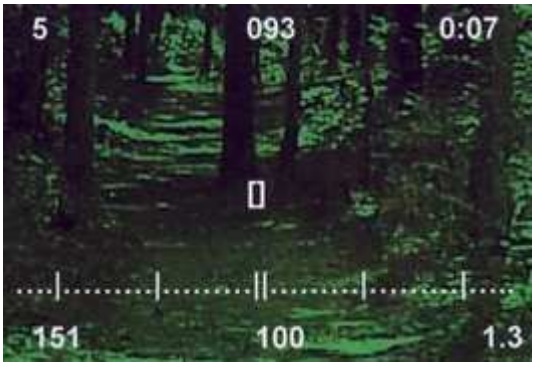


Figure 1: Current Text-Based HUD Displayed On NVGs [STS03b]

The HUD displayed in Figure 1 provides the following information:

- Next waypoint ID (top left)
- Bearing to the next waypoint (top center)
- Time to go until the next waypoint is reached (top right)
- Distance to the next waypoint (bottom left)
- Track made good (bottom center)
- Present ground speed (bottom right)
- Cross track error (center)

While this information is ideal for navigating on the ground, several critical pieces of information are missing for a parachutist. This includes altitude, rate of descent, and wind speed. Beyond the missing information, there is another, more fundamental problem with current HUD displays. The information they present is hard to grasp mentally. They focus on presenting data based on where the paratrooper is and the

direction he is moving. While there is some predictive data, such as the bearing to the next waypoint, ETA to the next waypoint, and distance to the next waypoint, the data is not formatted in an easily understandable format. The data is also limited in scope, as it only displays information on the next waypoint. If the paratrooper moves away from the path between two waypoints, he may have a hard time regaining the path. Also, the current HUD is geared for ground navigation – that is, movement in two dimensions. Paratroopers must work with three dimensions; their altitude is critical information. The current HUD does not adequately present information about the paratrooper's altitude in relation to his path, or how to correct the altitude to regain a path.

In addition, the current HUD does not use head tracking. This is vitally important for a paratrooper. The current HUD assumes that the user is looking in the same direction as he is heading. This assumption does not hold true for paratroopers. Crosswinds and headwinds may result in a paratrooper facing a direction completely different from his velocity vector. A HUD based on velocity measurements for heading will present a very confusing display to a paratrooper. A system for measuring the exact location and orientation of a paratrooper's head is necessary in order to develop a HUD suitable for paratroopers.

1.3. Research Focus

The focus of this research is to design, implement, and evaluate a prototype HUD displayed on NVGs, using data from GPS and an Inertial Measurement Unit (IMU) to enable a paratrooper to navigate in zero-visibility situations. A significant portion of this research is developing a graphical HUD that does not take up much computing time, to

enable it to run on low-powered, easily portable computers, and on how to compute an path to deliver a paratrooper from his release point to his landing zone. This research will also evaluate the effectiveness of the graphical HUD against conventional navigational cues.

1.4. Objectives

This research has two main objectives. The first objective is to design and implement a HUD to convey all the navigational information a paratrooper needs in order to execute a HAHO jump and land at his landing zone (LZ). The second objective is to evaluate the effectiveness of the graphical, predictive HUD in helping paratroopers navigate.

The first objective involves integrating data from a GPS (real-time user position), an IMU (real-time tracking of the user's head), and information about the jump (such as the LZ's location and the velocity of any local winds). This data is used to calculate a path for the paratrooper to follow in order to successfully land at the LZ. The path is depicted in an intuitive, three dimensional, graphical display, projected as a HUD on the paratrooper's NVGs.

The second objective involves testing the HUD software. While previous simulations have indicated that a graphical HUD is beneficial to navigating, it is unknown how well these simulations translate to the real world. A test comparing the graphical HUD and the textual HUD against a baseline source of navigational information validates the idea that a HUD is a useful navigational tool.

1.5. Assumptions

Several assumptions have been made in order to limit the scope of this research to a manageable level. First, it is assumed that this navigational information will only be used in a parachuting environment. Although the system may be adaptable for ground navigation, that is not the thrust of this research. Nevertheless, due to practical considerations, ground tests are used to evaluate the system. Second, it is assumed that data coming from the GPS and the IMU are perfectly accurate. Third, it is assumed that wind information used while generating the path is accurate. Fourth, it is assumed that parachute performance characteristics provided are accurate.

1.6. Approach

To achieve the objectives of this thesis, several steps were accomplished. First, an NVG HUD system was developed. This system uses a consistent process to generate the output displayed on the HUD. Information about the HAHO jump must be collected. This is done through the use of configuration files which the software will read. Necessary information includes the parachute characteristics, the location of the LZ, and wind information.

Once this information is collected, a release point and a path to the LZ are computed, based on the wind and parachute performance information. The release point is computed so that the distance from the LZ is maximized. This is necessary to ensure that the insertion can be made from the greatest possible stand-off distance. The path is displayed to the paratrooper as a series of boxes forming a virtual tunnel for him to fly through.

When the GPS/IMU system begins to generate information, the paratrooper's position and viewing angle is used to compute his location and attitude relative to the tunnel in the sky. This is used to update the display information. The GPS/IMU provides new location and heading information at up to 100 times a second. Every time a new set of data becomes available, the display is updated accordingly. This creates a realistic virtual world projected to the paratrooper, guiding him to the landing zone.

Once the system is created, the second step is that it must be tested to ensure that the concept works in the real world, and that the combination of the display technique and hardware work together. Because actual parachute jumps are prohibitively expensive and dangerous for this research, tests were conducted on the ground. A number of paths, defined by a set of waypoints, were generated. Test subjects then navigated a path using the baseline navigational aid (a GPS displaying a course pointer arrow) or the graphical HUD. The total time taken to navigate the course was the primary performance metric.

1.7. Document Overview

The remainder of this thesis is divided into five chapters. Chapter two explores previous research conducted into the field of using a HUD as a primary navigational reference, and other issues pertinent to this research. Chapter three details the implementation of the HUD software. Chapter four describes the methodology used in testing the system. Chapter five presents the test results and the analysis of the test data. Chapter six summarizes the research conclusions and presents topics for further development in this area.

2. Literature Review

2.1. Introduction

A number of factors must be considered when designing and implementing a primary navigation reference display for paratroopers. These areas include navigation by the Global Positioning System (GPS), helmet mounted displays, tunnel in the sky navigational displays, and parachute navigation.

2.2. Navigation with GPS

The Global Position System (GPS) is designed to provide highly accurate, three-dimensional position, velocity, and time information. This information is available anywhere on earth, twenty-four hours a day. The GPS system has cost over $14 billion dollars and has taken thirty years from its inception to being declared fully operational, but has proven to be worth the cost [PAG95].

Three main components make up the GPS system: the ground control segment, the space segment, and the user segment [ACK94]. All three parts must work in unison for the system to provide high accuracy.

The ground control segment consists of a number of stations located on earth. There are five stations around the world, each contributing to the maintenance of the GPS satellite constellation. The Master Control Station (MCS) is located at Schriever AFB, and functions as the central node in the GPS system. The MCS is run by the Air Force, and is in operation 24 hours a day, 7 days a week. It is augmented by four tracking and control stations around the world, as illustrated in Figure 2. The stations form a

"bracelet" around the earth of control centers allowing continuous monitoring of the satellites [ACK94].

Figure 2: GPS Ground Control Segment Locations [ACK94]

The MCS controls all aspects of the command and control of the satellite system. This includes controlling the movements of the satellites, estimating the navigational performance of the satellite signals, and calculating the ephemeris (the equations for the satellite's orbit, so that the exact position of the satellite can be calculated for any given time) data. Other stations, located at Hawaii, the Ascension Islands, Diego Garcia, and Kwajalein, track the positions of the satellites and collect data on the range of the satellites from the station. This data is sent to the MCS and used to refine the ephemeris data. In addition, the ground control stations (except for Hawaii) have uplink antennas

used to send commands and navigation data to the satellites [ACK94]. These uplink antennas are unmanned, and are under the control of the MCS.

The space segment consists of the satellite constellation, made up of at least 24 satellites, although 29 are currently active. The currently operational satellites are Block IIR, providing greater capability and longer service life than the previous Block I and Block II satellites. The Block IIR satellites have extra hardening against space radiation, more fuel capacity, and redundant processors on board. In addition, Block IIR satellites have the capability to autonomously navigate for up to 180 days, without any input from the MCS (though this will lead to degraded navigational performance). Block I satellites could only hold 3.5 days of navigational data [NAV03]. Block IIR satellites can also communicate between each other, to pass messages from the MCS to satellites that are not currently in view of the MCS.

The 29 satellites are in six orbital planes, approximately 20,200 kilometers above earth. The orbital planes are inclined at 55 degrees relative to the horizon. Each satellite orbits the earth approximately every 12 hours [ACK94]. These orbits were chosen so at least four satellites would be visible at all times from any point on the earth. Four satellites are the minimum number of satellites necessary for three-dimensional navigation, as will be explained below.

Each satellite is equipped with four atomic clocks – two cesium clocks and two rubidium clocks. These atomic clocks have a drift rate of less than $1x10^{-13}$ seconds – that is, they will gain or lose a second once every 316,000 years. This number is further improved by using the clocks to check each other. Rubidium clocks are less accurate

than cesium clocks over a long term, but on a short term, rubidium clocks are more accurate than cesium clocks. By using this fact, the precision of the clocks are greatly improved [ACK94]. Precise timekeeping is essential to the working of the GPS system.

Each satellite continuously transmits on two frequencies. One frequency carries what is called a C/A code (for Coarse/Acquisition) and the other frequency carries an encrypted P (precise) code. The C/A code is intended for the civilian receivers, but can be used by military receivers to help lock onto the P code. Both of the codes are made up of a stream of pseudorandom bits. The C/A code repeats at a rate of 1 kHz, while the P code repeats once a week. This is done to enhance the security of the P code; increasing the repeat period decreases the chances that an enemy will be able to acquire the P code [ACK94].

The data transmitted by the satellites are received by the user segment, or receivers on the ground. The user segment consists of cell phones, consumer GPS receivers, navigational equipment on aircraft, ships, and vehicles, and other pieces of equipment that are capable of receiving and deciphering the GPS signal. The user segment's location is variable. The ground control segment never moves; the satellites travel in carefully monitored orbits. The user segment's position changes, and must be determined for the GPS system to be useful.

In order to determine a location, the user's receiver determines its position from at least four satellites. A simple example helps illustrate how GPS works. Suppose Bob is completely lost somewhere in the United States. A local inhabitant of the area Bob is in tells him that he is exactly 540 miles from New York City. While this does not tell Bob

exactly where he is, he knows he is somewhere on the perimeter of a circle around New York City of 540 miles. Another inhabitant tells Bob that he is 450 miles from Memphis, Tennessee. This helps Bob tremendously. The circle around Memphis of 450 miles intersects the circle around New York City of 540 miles in exactly two places. Bob knows he is located at one of the intersections. A third inhabitant tells Bob that he is 590 miles from Minneapolis, Minnesota. This circle intersects the other two circles in exactly one spot. Bob knows he is at the intersection of these three circles (Wright Patterson AFB) [BRA04].

Figure 3: The Intersection of Three Circles Defines a Single Point

The GPS system works in a similar manner. The locations of the satellites are constantly monitored and very well defined. This information, called the ephemeris, is sent to the GPS satellites via the MCS, and from the GPS satellites to the receivers. The signal broadcast by the satellites moves at the speed of light. Bob's GPS receiver is able to determine how far it is from a GPS satellite by measuring the delay between when the satellite sent the signal, and when the receiver received the signal. By keeping the GPS

receiver's clock synchronized with the atomic clocks onboard the GPS satellites, the delay can be very precisely measured. Clock synchronization is performed by estimating clock error, using measurements from the satellites themselves.

GPS's popularity and practicality in the civilian world was greatly increased on 2 May 2000. On this date, Selective Availability (SA) was turned off [NAV03]. SA was a method used by the Department of Defense to deny precise location measurements to non-military GPS users. With SA turned on, the satellites would broadcast slightly incorrect time and ephemeris data [IGE00]. These intentional errors limited horizontal accuracy of a user's receiver to about 100 meters at the 95% confidence level. However, since the errors were introduced in a coded fashion, an authorized user with knowledge of the error-inducing method would be able to receive full GPS accuracy.

SA was continuously operating from 25 March 1990 to 1 May 2000, with the exception of from September 1990 – July 1991 (due to the first Gulf War). The DoD has stated that it does not intend to ever globally activate SA again. There is a capability to selectively turn on SA in certain regions of the globe, and this might occur in times of conflict.

Although the GPS system can be used for sub-centimeter measurements, commercial receivers intended for recreational civilian use are not this accurate. GPS receivers are more accurate measuring horizontal position than vertical position. This is due to geometry issues with the satellites. If the visible satellites are located close to each other, or in a straight line from each other, the error in the measurement of location grows. A measurement of this inaccuracy is called the Dilution of Precision (DOP).

DOP is unitless and can be broken down into Horizontal DOP (HDOP), Vertical DOP (VDOP), and so on [ACK94]. The DOP can be any integer greater than or equal to zero; the lower the number, the better the measurement.

2.2.1. Related Research in GPS Navigation

GPS receivers are being integrated into many different devices, including automobiles, cell phones, tracking devices, and so on. A fundamental concern, no matter what the device using the GPS receivers, is the accuracy of the position information being reported.

A typical advertised horizontal accuracy for a consumer GPS receiver is about 15 meters at the 95% confidence level [GAR05]. David Wilson conducted tests on a Garmin Etrex (a popular consumer-level GPS receiver), measuring its reported position against a surveyed point. Over the course of 48 hours, the Etrex's reported position averaged 3.0 meters away from the actual position, with the 95% confidence level at 6.7 meters [WIL05]. Consumer GPS receivers typically do not specify their vertical accuracy. However, they are not as accurate when reporting a user's altitude as they are when reporting their horizontal position. This is due to the satellites' geometry with the receivers on the ground, which usually ensures that VDOP is greater than HDOP. Wilson suggests that an accuracy of 20 meters at the 95% confidence level with a VDOP of 2.0 is likely for most consumer GPS receivers [WIL05].

Velocity is another key datum that is frequently obtained from the GPS, and plays an important role in the HUD system under development. Many commercial GPS receivers specify their velocity accuracy as within 0.1 m/s [GAR05]. [WIT04] conducted

testing of the accuracy of the velocity readings of a GPS receiver mounted on a bicyclist. The bicyclist's true speed was determined by using a very accurate speedometer mounted to the wheel of the bicycle. As the bicycle's wheel rotated, the number of revolutions was counted. By using the known circumference of the wheel and the time it took the wheel to make one revolution, the precise speed could be determined. The bicyclist rode in both straight and curved paths. It was found that the GPS receiver reported velocity correct to 0.1 m/s about 45% of the time [WIT04]. However, this was dependant on the path being ridden. The GPS-reported velocity was most accurate during straight line travel, as seen in Figure 4.

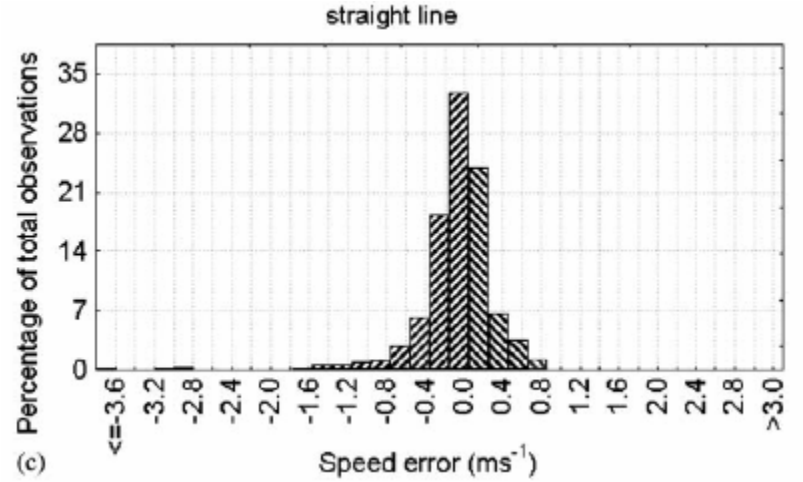


Figure 4: Speed Error For A Bicycle Traveling In A Straight Line [WIT04].

In Figure 4, a negative value on the graph indicates that the GPS was underreporting the speed; a positive value indicates that the GPS was reporting a speed greater than the actual speed. The GPS sensor was able to fairly accurately report the speed of the bicyclist, even when the bicyclist rapidly changed speed. Figure 5 illustrates the difference between the actual speed (white circles) and GPS speed (black circles).

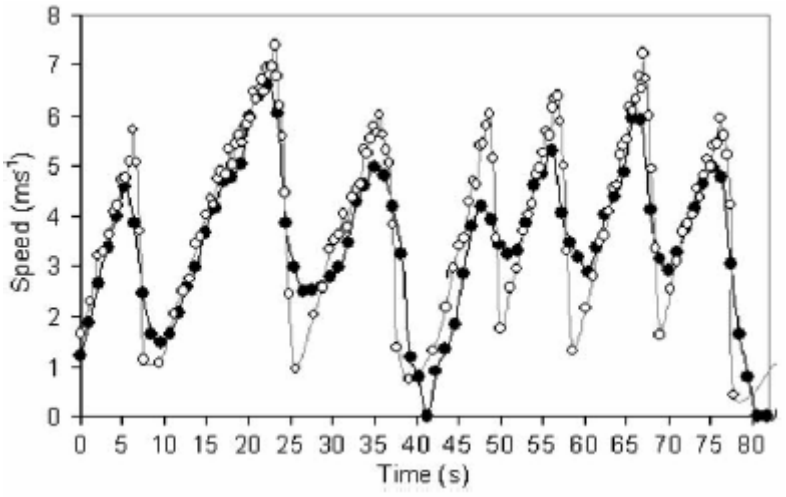


Figure 5: GPS Reported Speed Vs. Actual Speed For A Straight Line [WIT04]. Black Circles Represent GPS Speed; White Circles Represent Actual Speed.

While the bicycle was traveling in a circle, greater error was observed. Two types of circular paths were followed: a small circular path and a large circular path. The small circular path resulted in greater velocity errors than the large circular path, as seen in Figure 6 and Figure 7 [WIT04].

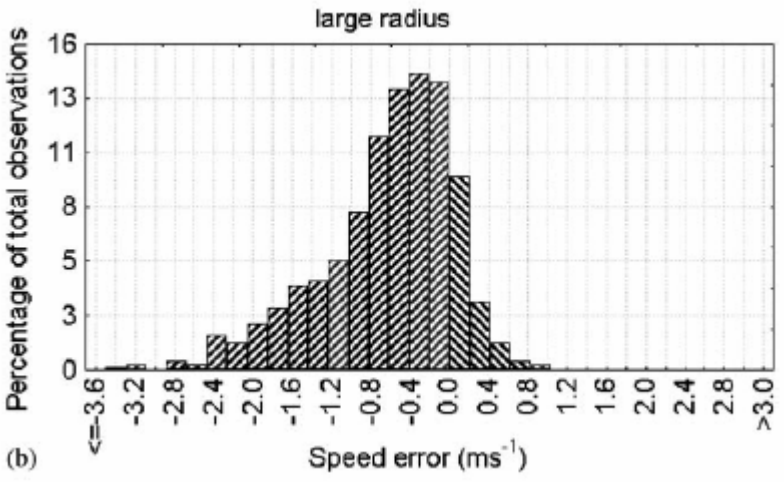


Figure 6: Speed Error For A Bicycle Traveling In A Large Circular Path [WIT04].

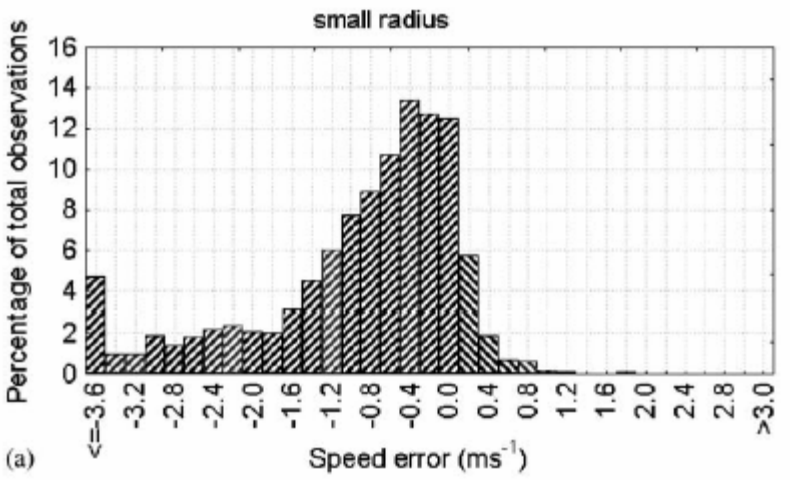


Figure 7: Speed Error For A Bicycle Traveling In A Small Circular Path [WIT04].

It is hypothesized that the source of these errors comes from the fact that the bicyclist was leaning into the turn. Since the GPS sensor was mounted on the bicyclist's head, his head traveled a shorter path than the wheels of the bicycle. In addition, by changing the GPS sensor's position relative to the satellites, the DOP of the fix was reduced in the turns [WIT04]. Another possible effect is the velocity algorithm employed by the GPS sensor. Although these algorithms vary between each manufacturer of GPS sensors, and are protected as trade secrets, it is known that some sort of Kalman filter is used. The tuning of the Kalman filter could affect the accuracy of the velocity reported [WIT04].

The inaccuracy of the GPS reported speed can be seen in Figure 8. Again, the actual speed is shown by the white circles, while the dark circles represent the GPS reported speed. By slowing down in the curves (indicated by the dark lines along the bottom of the graph), the bicyclist threw off the velocity reporting algorithm, which resulted in greatly under-estimated speeds.

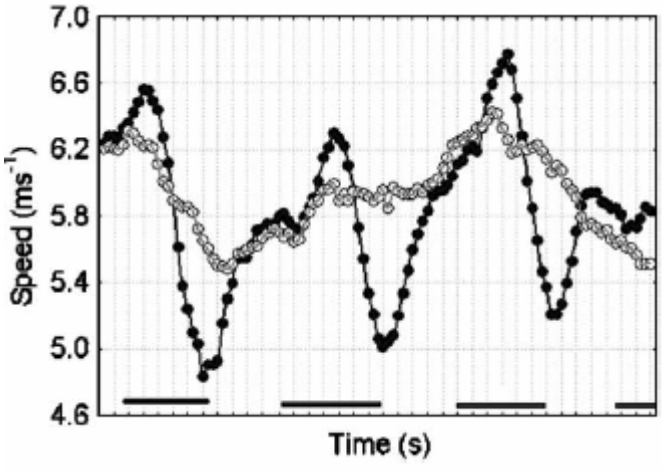


Figure 8: GPS Speed Vs Actual Speed For A Curved Path [WIT04].

2.3. Helmet Mounted Displays

A helmet mounted display (HMD) is a rapidly growing field of interest for conveying information to a person. A small display is mounted in front of the user's eyes so that the user can simultaneously view the display and his surrounding environments, similar to a conventional HUD. However, HMDs have the advantage that the information is always in front of the user's eyes, so that no matter where his head is oriented, he is able to view that data.

Some HMDs display information that is not sensitive to the user's head orientation. For instance, a pilot in an airplane may utilize a HMD displaying his plane's altitude, velocity, heading, and attitude. This information only changes when the plane's situation changes. It does not matter if the pilot is looking straight ahead, or to his left. Other applications do require tracking the movement of the user's head. For instance, if a pilot is using a HMD that places a symbol over an enemy airplane, the orientation of the pilot's head and the location of the enemy airplane become important. The tracking of the user's head is important, and can be accomplished in several ways.

Movement of the human body is measured according to some fixed reference point. This point can be either internal to the human body (for instance, rotation around a certain joint), or external (for instance, distance from a point in the cockpit). The displacement is measured by a sensor reacting to some source of data. This source of data can be either natural (for instance, gravity) or artificial (for instance, light emitted from an LED illuminating the scene). According to [MUL94], tracking the movement of the human body can be broken down into three taxonomies:

- Inside-in: The sensor is mounted on the human body, and senses a source coming from the human body. An example is a piezoelectric flex sensor mounted on a finger that measures how far it has bent.
- Inside-out: The sensor is mounted on the human body, and senses some external force to determine movement. A small accelerometer mounted on the user's body can measure its acceleration, and use that data to extrapolate the movement of the user.
- Outside-out: The sensor is located off of the human body, and senses a source from an external source. For instance, computer vision projects can use a camera to observe the subject, collecting the light reflected off of the subject's body, and calculate the position and orientation of the subject.

Each of these approaches to measuring head movement has advantages and disadvantages. Outside-out sensors tend to be limited in the space that they can monitor. If the user leaves the space the sensors are monitoring, then the system is unable to measure the person's movement. Outside-out sensors also tend to be, by far, the most

expensive systems to implement. However, the user is often free from wearing bulky measuring equipment. Inside-in sensors tend to be less expensive than Outside-out sensors and offer an unlimited workspace, but this is offset by the need to wear often intrusive measuring equipment. Inside-out systems fall in between the two other systems. They minimize the amount of equipment worn, and can offer an unlimited workspace (if sensing an artificial source). However, the Inside-out sensors often suffer from high latency, which can limit user acceptability [MUL94].

For head tracking to be acceptable to a user, it must meet certain performance criteria. These performance criteria include accuracy, repeatability, delay, and comfort levels.

Accuracy refers to the accuracy that the head tracking hardware is capable of. For instance, if the user turns his head 5 degrees, the head tracking hardware should accurately reflect that. If the hardware reports that the user turned his head 20 degrees, the difference in position will rapidly make whatever data the head tracker is generating, useless. Repeatability is related to accuracy. If the user of the system moves in a certain manner, the system should measure the same distance, each time.

Delay is defined as the difference in time from when the user turns his head and when the system updates the user's display. The latency consists of time for the measurement device to detect movement, calculate the distance and direction moved, transfer the data to the software updating the HMD, and the HMD updating. Obviously, the shorter the time interval between the head movement and the HMD update, the better. Reducing latency is especially critical in applications where the HMD is painted over real

world objects. The condition where the HMD screen element (say, a target indicator) is catching up to the real world object (the target) is known as swimming. Researchers disagree on the maximum limit of lag that can be tolerated by users. [NEW00] recommends a limit of no more than 20 milliseconds between movement and the HMD update, while [RAS98] found that acceptable limits could range anywhere from 40 ms to more than 300 ms. No matter what was recommended as an acceptable upper limit for delay, all of the studies found a direct relationship between the delay and the time it took a subject to complete a certain task, such as designating a target [RAS98]. In addition, increased lag was found to contribute to motion sickness in the user [RAS98].

2.3.1. Related Research in Helmet Mounted Displays

Tracking the head's position is vital in HMD applications. One Inside-out approach to head tracking is to place an inertial measurement unit (IMU) on the user's head or helmet. The IMU can track both the rate of acceleration and rotation around three axes of the helmet, and this information can be used to determine the head's location and orientation. However, the smaller the IMU, the less accurate it is, as the accelerometers have less mass to measure.

[JOF03] conducted tests using a small, inexpensive IMU to measure the position of a person's head. The IMU fed data about its acceleration and rotation to a system at a rate of 100 Hz. To account for the drift inherent in the IMU, a Kalman filter was employed. The filter used updates from a GPS at the rate of 1 Hz and estimates of the error present in the measurements to generate new estimates of the error present. By using these estimates as feedback to new measurements coming in from the IMU, the

Kalman filter could continuously 'calibrate' the IMU and minimize drift. Before the Kalman filter was implemented, the roll, pitch, and yaw drift rates of the IMU were 1.3°, 5.6°, and 1.3°, respectively, over a course of 60 seconds. After the Kalman filter was implemented, the IMU roll and pitch drift rates were found to be 0.068° and 0.48° over a course of 60 seconds. However, the Kalman filter was not effective in correcting for the yaw's drift rate [JOF03]. The Kalman filter could also be applied to the IMU's measurements of position and velocity with good results, as seen in Figure 9 and Figure 10.

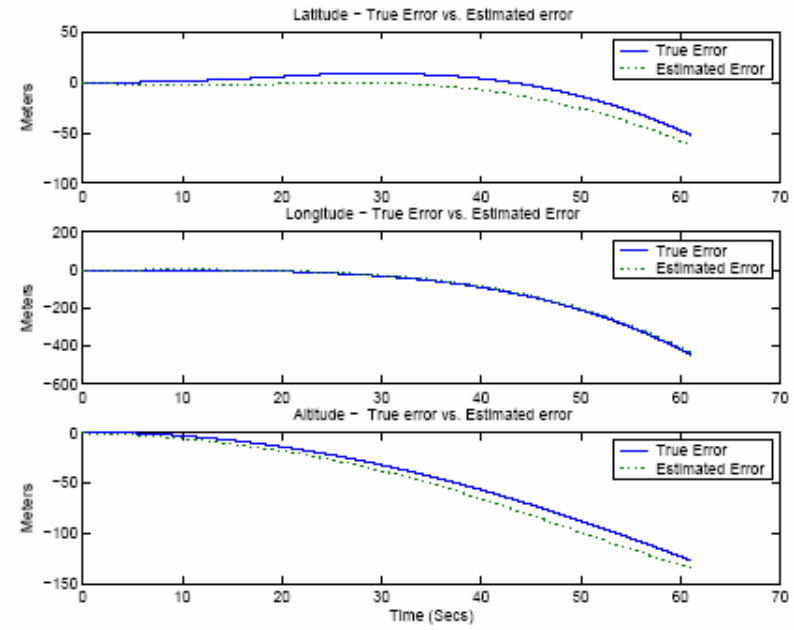


Figure 9: Actual and Estimated Errors in Position of the IMU [JOF03].

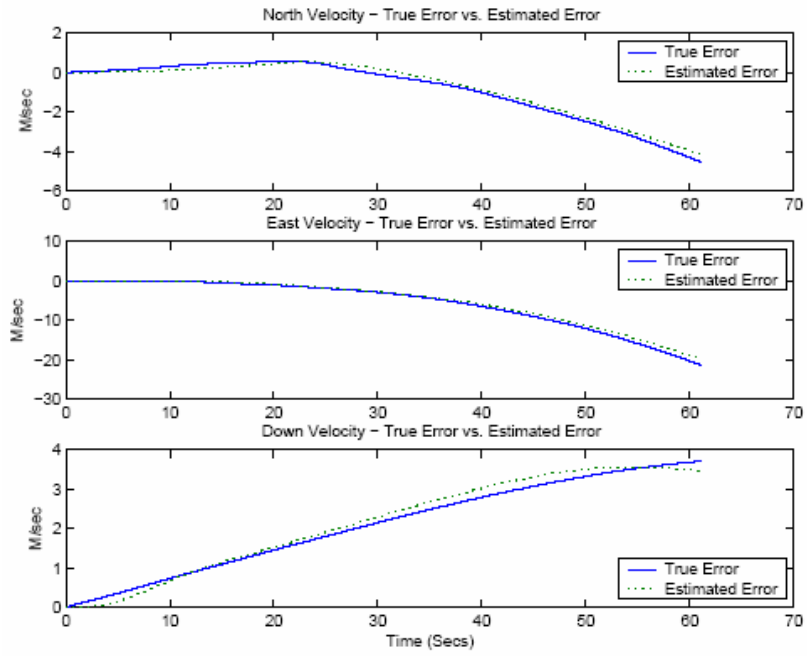


Figure 10: Actual and Estimated Error in Velocity of the IMU [JOF03].

Without the Kalman filter, the INS's reported position would drift away from its actual position by up to 430 meters over a 60 second period; with the Kalman filter, the reported position only drifted 5-10 meters over the 60 second period. This figure is expected to be improved with further filter tuning [JOF03].

The delay in the system between receiving data and outputting the system's orientation was not specifically described. However, it was noted in the paper that receiving the GPS data introduced a latency of 40 ms, due to the speed at which data came in from the serial port [JOF03]. Therefore, there is at least a 40 ms lag between receiving data and updating the delay. [RAS98] indicated that 40 ms is at the faster end of the range of acceptable lag. Further research should be conducted to determine the actual delay.

2.4. Parachute Navigation

Parachutes have drastically evolved since they were first used in combat during World War II. While previous generations of round parachutes offered relatively gentle landings, they possessed little to no inherent navigational capabilities. They floated straight down and in the direction of the prevailing winds. However, the development of ram-air parachutes changed the relationship between parachutes and the wind.

Ram air parachutes are essentially semi-rigid wings that are capable of creating lift and slicing through the air at speeds up to 60 miles per hour. Ram air parachutes are made up of a number of 'cells,' or pockets of fabric sewn together. Seven or nine cells are the most common parachute configurations. These cells inflate with air when air flows over and through the airfoil. The internal pressure of the cells keep the parachute semi-rigid; the faster the airflow over the airfoil, the more rigid the airfoil becomes. If there is not enough airspeed, then the parachute is at risk of collapsing. The ram-air parachutes do not produce thrust in order to gain their forward speed. Gravity accelerates them and causes them to move forward. The same effect causes a sled to speed down a snowy hill; the sled does not produce thrust, but gains forward speed because of gravity [SOB94].

Ram air parachutes can also steer themselves. The parachutist, by pulling on cables connected to the parachute (toggles), can warp or bend the wing. This warping causes a change in lift created on each side of the parachute, and the parachute turns. Pulling on the toggles too much can lower the amount of airflow over the parachute, which can cause the parachute to collapse. This is undesirable. Sharp turns, because the

amount of lift created is drastically reduced, cause the rate of descent to greatly increase. Depending on the circumstances, this could be desirable, but the parachutist must keep this fact in mind [SOB94].

A parachute's sink rate is affected by its forward speed. At greater forward speeds, the parachute will sink faster, and vice versa. However, this rate is not linear. The highest sink rates occur when the parachute has stalled (or has no forward speed) or when the parachute is in a dive (and has the highest sink rate). The rate of turn also affects the sink rate [STS03a].

The physics behind navigating a ram-air parachute are very similar to the physics behind paddling a canoe across a river. Where a canoe is affected by the velocity of the current of the river, a parachute is affected by the velocity of the wind. Ground-speed describes the speed of the parachute relative to the ground. Airspeed describes the speed of the parachute relative to the air. The ground-track is which heading the parachute is taking, relative to the ground [GER04].

The parachute behaves as if it is attached to the air. If the parachute is capable of 10 meters per second forward airspeed, and it flies directly into a wind which is blowing at 10 meters per second, the parachute's groundspeed will be zero meters per second. If the parachute turns and flies with the wind, it will have a groundspeed of 20 meters per second.

A parachute can 'crab' with the wind. This is done by flying at an angle to the wind. When the parachutist does this, their ground track will be a sum of the vectors of

the parachute's forward velocity and of the wind's velocity. Figure 11 illustrates how a parachute's ground track is affected by the parachute's heading.

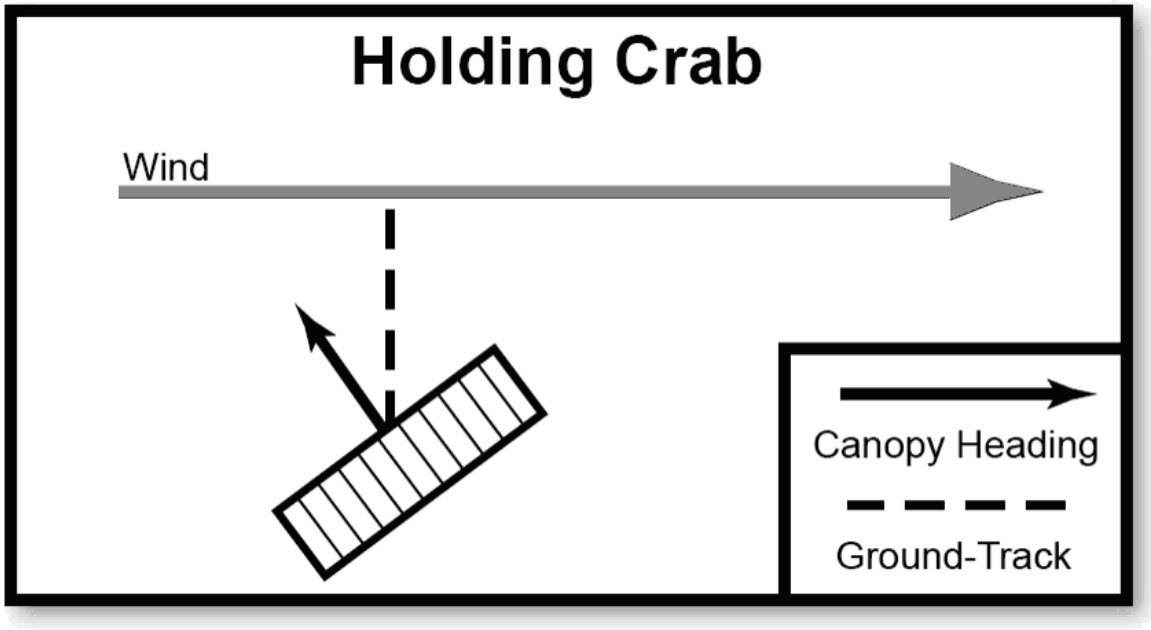


Figure 11: Holding Crab [GER04].

2.4.1. Related Research in Parachute Navigation

This thesis builds on the work of [BAL03], which explored the implementation and design of a three dimensional primary flight reference for paratroopers, displayed on a HMD, and integrated with NVGs.

Although the project did not create an operational model, a prototype simulation of the HUD was created and evaluated. [BAL03] found that users were able to stay relatively close to the optimal path, averaging 2.15m away from the optimal path. This is within the horizontal distance of the tunnel edge from the optimal path, which indicates that the creation of a well-defined tunnel helps indicate to the user exactly where to fly. Also, there were a few instances where a user strayed far from the optimal path, but was able to fly back to the path and continue.

However, the research had limitations. The simulation was conducted on a PC, with the user sending inputs to the system via the arrow keys on the keyboard. The user could only control their heading, and could not vary their forward speed or rate of descent. The simulation was thus fairly artificial, and the results can't be extrapolated too far into the real world.

An operational parachuting navigation system exists, created by SSK Industries of Lebanon, OH [SSK05]. The system, termed OPANAS (Operational Paratroopers Navigation System) utilizes a small LCD screen and floating compass mounted to the paratrooper's chest to display navigational information. OPANAS is designed to help guide a paratrooper along a set of pre-defined waypoints to a landing zone. The system uses GPS signals to determine the paratrooper's location, a barometer to determine altitude, and magnetometer to determine heading. The navigational information generated by OPANAS, including the user's heading, range to the LZ, bearing to the LZ, and altitude, is displayed on an LCD in front of the paratrooper, as seen in Figure 12.

Figure 12: Photograph Of The OPANAS System In Flight [SSK05].

The OPANAS system does have some weaknesses. First, it is not designed to generate a path on its own. The system must be attached to a laptop in order to receive path information (though the laptop is removed before the jump). Information about the parachute's characteristics, winds aloft information, the desired landing zone, and all intermediate waypoints must be entered into the laptop, and then transferred into the OPANAS system. Also, the system must be opened and activated after the parachute has been deployed – it cannot be worn and ready to go during the freefall portion of the jump [SSK05]. The system also presents a top-down view of the paratrooper's intended path. The paratrooper must then translate one frame of reference (a top down view) to a second frame of reference (his vision). This may lead to difficulty in stressful situations. Early Apache helicopter HMDs used a similar overlaid map on the display, where the pilot had to convert from one coordinate frame system to another. This was found to increase the chances of the pilot becoming spatially disoriented [NEW00]. However, the OPANAS system is operational, and has been fielded with several NATO armies, and the Singapore Army Special Forces [SSK05].

Other forays into guiding parachutes examine the feasibility of autonomously guiding a parachute using small servos controlled by a computer for steering, based on inputs from the GPS. Of particular interest is the ability to airdrop cargo accurately from an aircraft, and have the cargo fly itself to an appropriate landing zone. The guidance, navigation, and control (GNC) algorithms for these systems have been explored for the past ten years [KAM03]. One recent example is the Pegasus GNC algorithm [KAM03].

The Pegasus GNC algorithm is designed to guide a 650 sq. ft. parafoil with a 500 lb. payload from a given release point to a landing zone. Inputs to the algorithm are the wind conditions at the landing zone, the release point, and the landing zone. The system can then autonomously fly from the release point to the landing zone by actuating servos in response to GPS signal inputs.

The Pegasus GNC takes a unique approach by computing a real-time path for the parachute. Using GPS inputs to determine its location and inertial path, the Pegasus GNC treats the wind as a disturbance. Since the path is computed in real time, the system can react to changing wind conditions. This is an advantage, as previous studies found that errors in predicting the wind conditions contributed the most to missing the landing zone [KAM03].

The Pegasus, and most other autonomous parachute systems, optimizes its flight path for safety rather than distance. By dropping the parachutes closer than necessary, the parachutes have an excess of potential energy. The flight path is thus designed so that the parachute flies in circles, burning off some of this potential energy. The energy-dissipating portion of the flight can take place either in the beginning, middle, or end of the flight path (Figure 13). Where the potential energy management maneuvers take place is dictated by the tactical situation, but does not affect the overall performance of the system. The only constraint is that the parachute must still have enough energy to land facing the wind at the LZ.

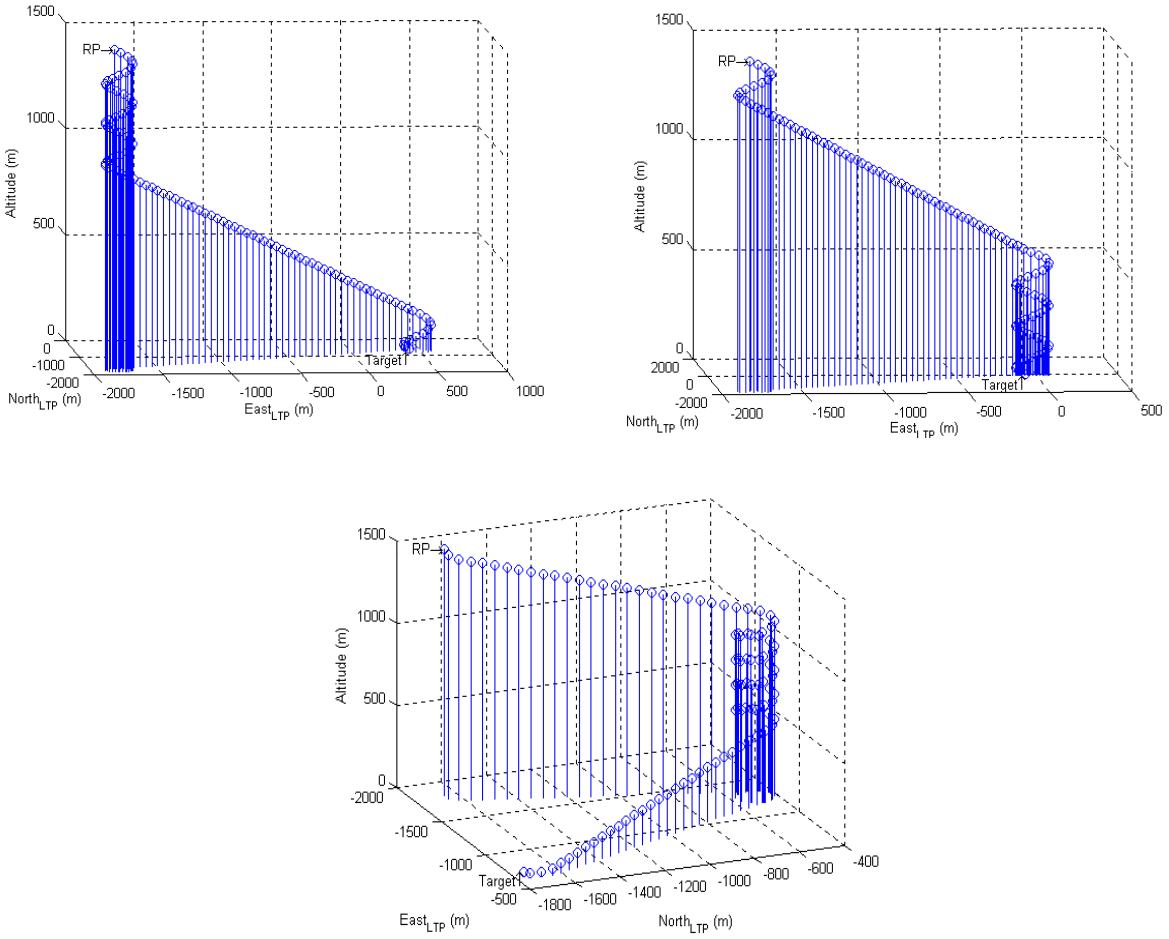


Figure 13: Early, Late, And Middle Potential Energy Management Paths For The Pegasus GNC Algorithm [KAM03].

Simulation tests of the Pegasus GNC indicate that it is fairly successful at autonomously landing a parachute. [KAM03] modeled GPS error (both position and velocity) as white noise, and used real-life wind data in their simulations. They found that the Pegasus system, on average, landed within 67.5 meters of the intended LZ, with a standard deviation of 20.5 meters.

2.5. Tunnel in The Sky Navigation Displays

Every modern aircraft has some sort of navigation display that indicates to the pilot information about where he is, and where he needs to go, though the exact layout and location of the display may vary.

MIL-STD-1787 defines the standard military Heads Up Display (HUD) [SNO099]. The primary focus of the MIL-STD-1787 HUD is to convey flight altitude information, such as the pitch, altitude, and airspeed of the aircraft. However, the HUD also indicates information such as the bearing to the next waypoint, a course indicator, a vertical deviation indicator, and so on. Figure 14 is an example of the MIL-STD-1787 HUD. The USAF feels that the MIL-STD-1787 HUD conveys sufficient information to act as a Primary Flight Display (PFD), offering enough information to conduct instrument-only flights [SNO99].

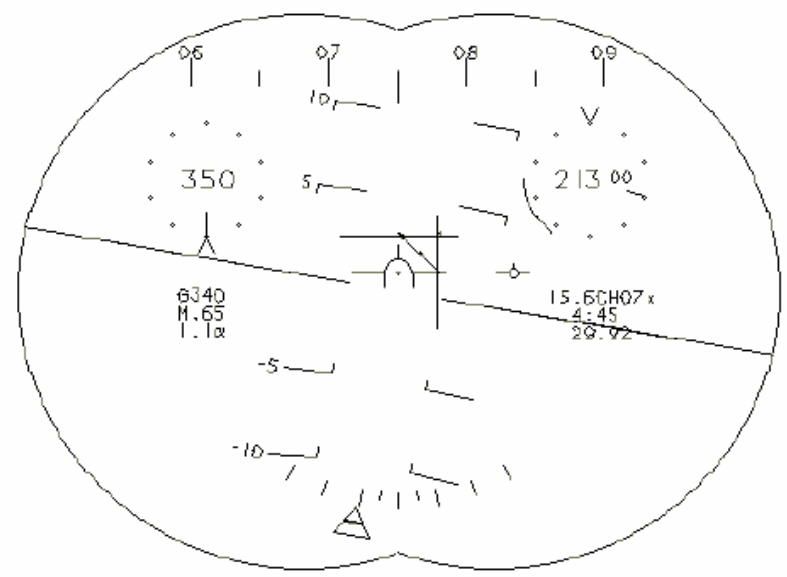


Figure 14: MIL-STD-1787 HUD [NEW00].

This type of HUD, while useful for conveying flight altitude information, is hard to navigate by. The MIL-STD-1787 HUD is based on gauges and instruments that have been present in the cockpit since World War II. Extrapolating from current design concepts leads to further complexity; this is undesirable and will not scale well as air traffic density increases [PRI03]. Commercial airliners often fly in situations, such as landing patterns, which require them to follow precise flight profiles in order to avoid collisions with other aircraft. Because the MIL-STD-1787 HUD is inadequate for this type of flight, other methods to display navigational information have been developed. One promising technique is the Pathway in the Sky method.

Interest in the Pathway in the Sky method grew during the mid-1990's, as computing power and computer graphics complexity increased. An area of interest to NASA was developing a Synthetic Vision System (SVS), where a representation of the real world could be projected to the pilots of an aircraft or spacecraft. This had the advantage of eliminating cockpit windows from the design of an aircraft/spacecraft [PAR03]. NASA began to research how different representations of the real world, and of the path the pilot of the craft ought to take, affected the accuracy and workload of the pilots.

Several different techniques began to manifest themselves. Some were two-dimensional in nature. For instance, one display used a grid of squares, with one square highlighted [NEW00]. By centering the highlighted square in the grid, the plane would be kept on course. More compelling displays used three-dimensional computer graphics to display a path to follow. Many different styles of these pathways in the sky were

developed, including flat paths, a 'follow-me' aircraft, and tunnels [PAR03]. Tunnels were found to be the most intuitive for pilots [PAR03].

Several methodologies for building the tunnels have been developed, ranging from the minimalist to the complex. Some configurations use a flat series of rectangles, creating a pathway resembling a sidewalk suspended in the air. These configurations create a flat path, and are known as pathways in the sky [SNO99]. Other configurations use rectangles to define a tunnel around the path that should be taken by the pilot. These configurations are known as tunnels in the sky [PAR03]. Additional information can be incorporated into either of these methods by the use of virtual signposts attached to the path, indicating the time or altitude that the pilot should be at when passing over or through that particular segment of the path.

Different methods of anchoring the tunnel to the real world have also been investigated. Most tunnels remain static and tied to the real world. Some efforts have been made to investigate 'noodle' tunnels, where the ends of the tunnel are fixed, and the path itself can move. One end of the tunnel is fixed to the desired destination, and the other end is fixed to the pilot's location. The tunnel changes shape and position, showing the pilot where to fly in order to get back onto the optimal path. However, 'noodle' tunnels were found to be distracting and annoying. Pilots did not like the 'whipping around' effect of the noodle tunnels [PAR03].

2.5.1. Related Research in Tunnel in the Sky Navigation Displays

Much of the research related to Tunnel in the Sky navigation displays seek to determine which tunnel layout results in the least deviation from the desired path.

[PRI03] tested four different tunnel layouts against each other and the MIL-STD-1787 HUD in navigating a complex course. Several experienced pilots were each randomly assigned a flight path, a display symbol set, and starting position. Their task was to fly the indicated flight path and land at an airport.

Four different tunnel concepts were tested. Every tunnel was made up of a series of rectangles surrounding the path to be followed, but differed in the way the rectangle was displayed. The first was the minimal tunnel, where the tunnel was defined only by small markers (a 'crow's foot' with the toes pointing along the edges of the rectangle, and the direction of the path) at the edge of each rectangle. The opposite approach was the box display. In this display, each rectangle was fully drawn, and connected to each other rectangle. A third approach was called 'dynamic crow's feet.' When the pilot is on course, the crow's feet defining the edges of the tunnel are kept small. However, the further the pilot strayed from the ideal path, the longer the toes of the crow's feet grew. The idea is to keep the display uncluttered if the pilot is on course, but provide more feedback if he is off course. The fourth tunnel in the sky configuration tested was a dynamic pathway. This is similar to the dynamic crow's feet, but the bottom of the tunnel is always drawn. The other toes of the crow's feet grow larger as the pilot strays further from the optimal path. The effect is a 'trough' guiding the pilot. These different configurations can be seen in Figure 15.

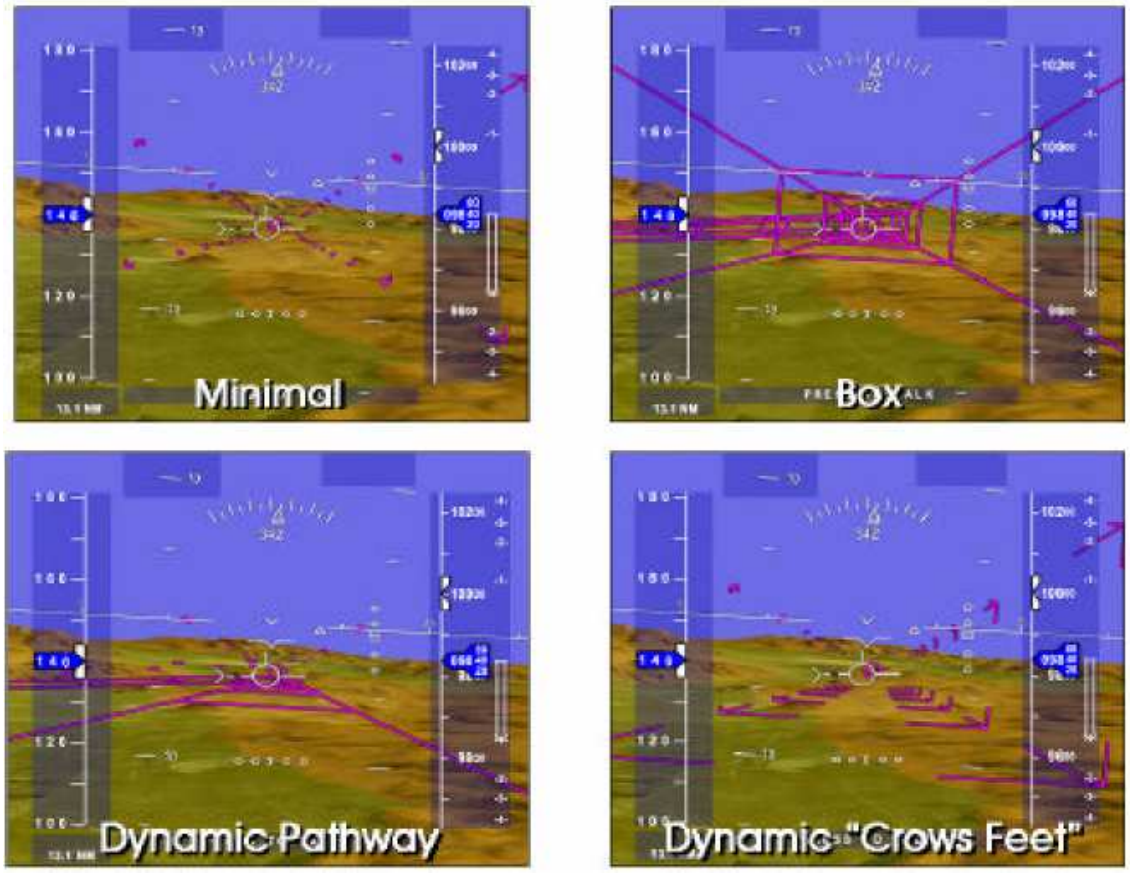


Figure 15: Different Tunnel in the Sky Configurations Tested By [PRI03].

The four Tunnel in the Sky configurations were tested against each other and the MIL-STD-1787 HUD in a flight simulator. The testing found that with the tunnels, no matter the configuration, the pilots were able to follow the path much more accurately and have better situational awareness than the standard HUD layout. The tunnels also significantly reduced the workload on the pilot. Pilots tended to prefer the dynamic crow's feet tunnel. While there was not much difference in performance between the various tunnel layouts, the minimal tunnel did tend to perform slightly worse in keeping the pilots on track than the others. However, even the minimal tunnel significantly outperformed the standard HUD [PRI03].

The testing conducted by [PRI03] validates the idea of a tunnel in the sky in clear weather. Other research indicates the same concept can be applied in varying visibility conditions. [SNO99] used a flight simulator flown by military pilots to gauge the effectiveness of a pathway in the sky (a flat path rather than a tunnel) versus the MIL-STD-1787 HUD for guiding pilots in inclement weather conditions, where visibility was severely restricted. Pilots had to make a landing at an airfield given an 'easy' or a 'hard' approach. Wind could be present and disturb the path of the aircraft.

[SNO99] found that pilots using the pathway in the sky HUD were able to land with the same accuracy and maintain a commanded airspeed equally well, regardless of visibility conditions. Pilots preferred the pathway in the sky technique over the standard military HUD, as well, claiming that the pathway increased situational awareness, even when the ground was not visible.

2.6. Summary

This thesis builds upon four areas of knowledge: navigation with GPS, helmet mounted displays, tunnel in the sky navigational displays, and parachute navigation.

The GPS system uses a constellation of satellites to determine a receiver's position anywhere on the face of the earth. A network of ground control stations around the world monitor and control the satellites. A receiver on the ground can monitor signals from the satellites. By tracking at least four satellites and determining its position from each of them, a receiver can determine its exact location. Commercial GPS sensors can be expected to have a horizontal accuracy of about 7 meters at the 95% confidence level and a vertical accuracy of about 20 meters at the 95% confidence level.

HMDs are small displays mounted over the user's eyes. HMDs frequently utilize the orientation of the user's head to enhance their usability. There are several methods for tracking the orientation of the user's head, including Inside-in, Inside-out, and Outside-out techniques. Each of these techniques has advantages and drawbacks. Head tracking must meet several performance criteria in order to be useful for a HMD application. Delay, accuracy, and physical comfort are some of these criteria.

Parachutes have evolved from simple round parachutes that fell relatively straight down to high performance ram-air parachutes. These ram-air parachutes are essentially semi-rigid airfoils and are capable of significant forward speed and navigational capability. However, parachutes are affected by the wind, and behave as if they are attached to the wind. A strong headwind could result in a parachutist flying backwards. Parachutists must sometimes crab into the wind in order to fly where they wish to.

Tunnel in the Sky navigational displays grew out of an increasing need for intuitive, useful navigational cues for pilots, and by an increase in computing power. These two factors led to the development of virtual tunnels in the sky for a pilot could fly through in order to follow a path. Testing has shown that pilots are able to more accurately follow a path while following a tunnel in the sky versus the standard military HUD. In addition, this improvement in accuracy is maintained even during periods of limited visibility, such as during inclement weather.

3. System Description

3.1. Introduction

This chapter discusses the design and implementation of the software for a paratrooper's primary navigational reference, presented on a NVG/HUD. The overall design of the system is first described, with emphasis on the Allegro graphics library [HAR05]. Then, a detailed discussion of the implementation of the design is given. In addition, the HUD displayed to the user will be explained, as will the hardware used to display the HUD.

3.2. Coordinate Systems

The development of the software uses three separate coordinate systems. These coordinate systems each describe a separate world. The worlds are the real world, the virtual world, and the screen space. The three worlds are illustrated in Figure 16.

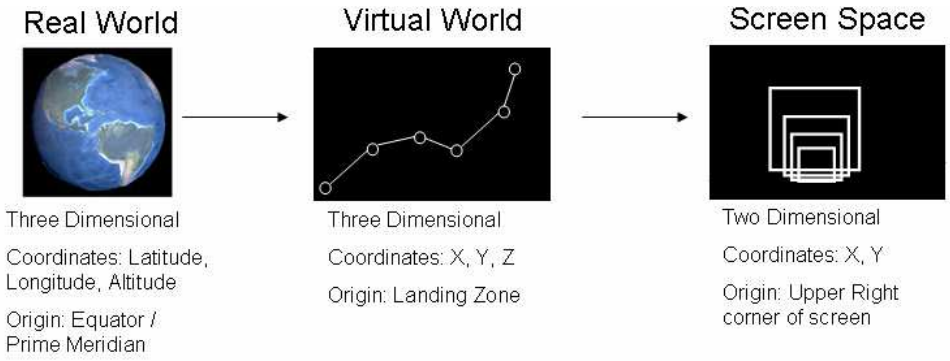


Figure 16: The Three Worlds That Play A Role In The HUD System

The real world is the world that physically exists. A point on the earth can be described by three parameters: latitude, longitude, and altitude. The user inputs the LZ's location in the real world to the system. The system then defines a virtual world using units that are more convenient for the system to handle, converting the real world

coordinates to the virtual world's coordinates. The virtual world's origin is at the LZ, or (0, 0, 0). The specifics of the virtual world's coordinate system are explained in more detail in Section 3.5.1.

Once the virtual world is generated, the system has a three dimensional representation of the situation. These three dimensions must be translated into a two dimensional image on the screen (i.e., into screen-space). The screen-space image is what is finally presented to the user. The translation from three-dimensional virtual world coordinates to two-dimensional screen-space coordinates is handled primarily by the software package used to develop the HUD.

3.3. Software Design

The HUD developed for this thesis was developed using a graphics library called Allegro, and written in C. The other key component of the HMD is the software supplying the user's position and attitude information to the HUD software. Both of these programs run concurrently on a single computer.

3.3.1. Description of Allegro

Allegro is a cross-platform multimedia library originally intended for game programming on DOS platforms. Over the years, it has evolved into a cross-platform, rich multimedia programming library [HAR05]. It has many features making it suitable for graphical programming, especially on lower powered computer hardware, since it does not require advanced hardware.

Allegro's strength lies in its substantial multimedia capabilities. Allegro's libraries include functions for playing music and sound effects, routines for interfacing

with the keyboard, mouse, and joystick, a Graphical User Interface (GUI) development library, and other features necessary to develop a game. The implementation of this thesis uses only a subset of Allegro's libraries.

The first of component used are the three dimensional mathematical routines. The three dimensional math routines are heavily used in generating the HUD. The tunnel and waypoints displayed in the HUD are parts of a virtual world, with each point on an object represented by a matrix. This matrix contains the three-dimensional coordinates of the point, and thus requires a 3x3 matrix. However, in order to transform the three-dimensional coordinate to the two-dimensional screen of the HMD, a fourth column and row is used in order to simplify the mathematics and increase the efficiency of the graphics library [HAR05]. Allegro abstracts all of this into a number of very simple function calls, allowing the programmer to focus on programming, rather than mathematics.

Another heavily used portion of Allegro's libraries is the primitive drawing routines. Any computer generated scene is composed of a number of "primitives," such as single pixels, lines, ellipses, and polygons, combined to form a complete scene. Analogously, the Mona Lisa is composed of a number of single brushstrokes all combined together to form the final image. Allegro abstracts the complex steps of writing to the video memory into simple commands, which speeds development time.

A major attribute of Allegro that makes it so attractive to this thesis is its cross-platform capability. Allegro is very portable, with most code being able to compile and run without modification on DOS, Windows, through the Linux console, through Unix's

X-Windows system, MacOS, and even some obscure operating systems like BeOS and QNX [HAR05]. The development of the HUD is intended for a low-powered machine running Linux. However, if a more suitable choice of hardware and operating system presents itself in the future, it is likely that the source code will only need to be recompiled in order to adapt to the new hardware.

Allegro is described by the creators as "gift-ware." As the documentation states, Allegro was "created by a number of people working in cooperation, and is given to you freely as a gift." [HAR05]. The creators of Allegro impose no license or restrictions on the use of Allegro. The code is free to be used for any purpose.

3.3.2. Description of the Information Supplying Software

While Allegro is responsible for generating the HUD, data is required in order to generate the graphics. This data is generated by software that manages the positioning hardware, and then captures, filters, and presents data from the hardware to the HUD.

From the point of view of the HUD software, the exact type of positioning hardware is irrelevant. The Information Supplying Software (ISS) is responsible for managing the positioning hardware and presenting data to the HUD, but is completely separate from the HUD software. The data the ISS is responsible for supplying area timestamp; the current latitude, longitude, and altitude; heading/orientation information, velocity information, and a data validity checksum.

Although the data the ISS generates is necessary for the HUD software to work, the details of the ISS are outside of the scope of this thesis, and so they won't be discussed in detail. From the HUD's perspective, the details of the ISS do not matter, as

long as it supplies the right data. For more details on a possible implementation, see [JOF04].

3.4. Sequence of Actions

The software developed for this thesis accomplishes a certain task – guiding a paratrooper to an LZ. In order to do this, the software must execute a number of steps. These steps are depicted as a flow chart in Figure 17. The steps can be broken down into a number of groups: gathering information, calculating the user's position and orientation, and generating the virtual world and HUD.

The first step to take in the execution of the program involves gathering information and parameters about the jump. There are three primary pieces of information that must be collected:

- Landing zone information
- Parachute performance characteristics
- Wind information

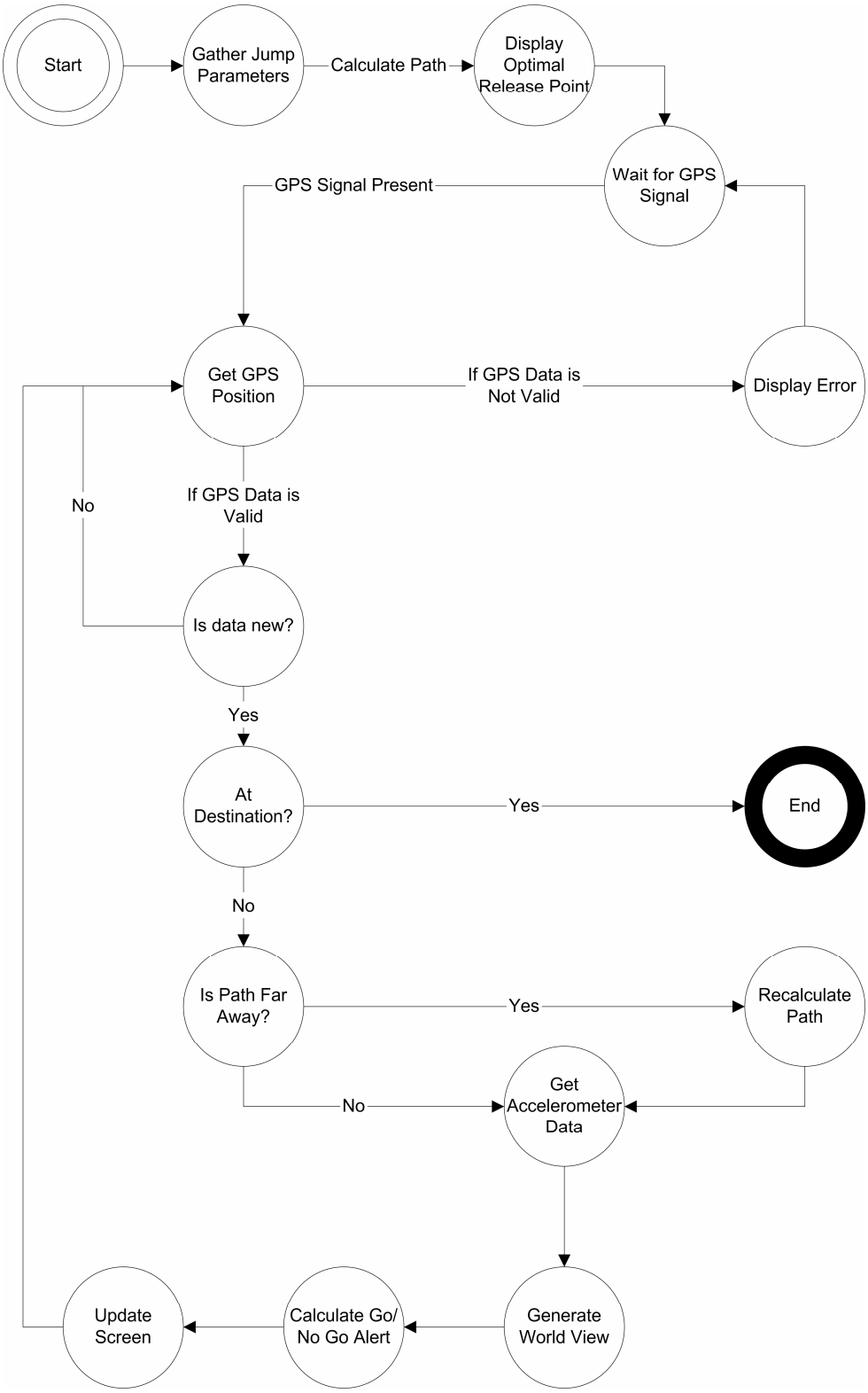


Figure 17: Sequence Of Events During Program Execution

Without this data, it is impossible to generate a flight path. Pertinent parachute performance parameters include its maximum forward velocity, its rate of descent, and its maximum and minimum opening altitudes. Information about the landing zone consists of its latitude, longitude, and altitude. Wind information is broken up into several altitude increments, starting from ground level, and going up to the maximum opening altitude of the parachute. A wind direction (defined as the direction the wind is blowing *from*) and wind velocity is assigned to each altitude increment. Wind information can be collected via two means – either by weather forecasts, or by the aircraft carrying the paratroopers. The aircraft overflies the target location, then begins a climb to the release altitude. As the plane climbs, it measures the wind speed and direction [STS03a]. The wind information should be obtained by measurement, because the more accurate the information is, the easier it will be for the paratrooper to follow the generated path.

Once the information has been gathered, the path can be calculated, and the ideal release point is displayed to the paratrooper. Once the release point has been computed, the system waits for data from the ISS to arrive.

If the data received is valid, then the system calculates the paratrooper's position relative to the LZ. If the paratrooper is 'close' (within 15 seconds, given his current rate of travel) to the LZ, then the program terminates. If the paratrooper is 'far away' from the generated path (if, for instance, the plane did not release him in the proper place, or if he has drifted far away from the originally generated path for some reason), the system recalculates an optimal path from his current position. Whether or not a paratrooper is 'far away' from the path is determined by calculating the paratrooper's distance from his

next waypoint and the LZ. If the paratrooper's distance to the next waypoint is greater than the distance between his previous waypoint and his next waypoint, and he is further away from the LZ than his next waypoint, then the paratrooper is 'far away.' This algorithm is illustrated in Figure 18.

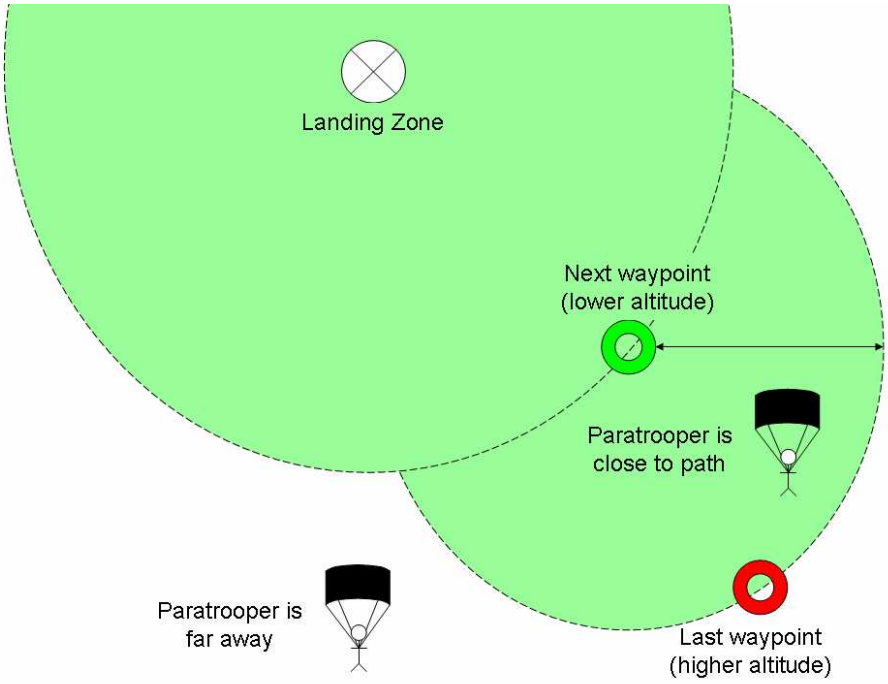


Figure 18: 'Close' And 'Far Away' Areas (Areas Shaded Green Are 'Close').

Once the paratrooper's position relative to the LZ and the path is known, the system generates an appropriate display. The display includes the view of the path if it is visible, and if it is not visible, a "guide arrow" indicating which direction the paratrooper needs to look in order to bring the path back into view. Other elements of the display include the paratrooper's altitude, ground speed, and heading. In addition, the system will determine if the paratrooper is making enough progress towards the LZ in order to successfully arrive at it. If the paratrooper is falling too quickly or too slowly, the system will alert him to this fact by drawing a red border around the HUD. Once all of this has

been generated, the screen will update, and the system will wait until a new GPS/IMU dataset arrives. This process repeats until the paratrooper arrives at the landing zone.

In addition to the HUD software, the PC-104 will be concurrently running the ISS software which reads and filters data from the GPS/IMU system. Therefore, the HUD software must not consume too much processor time, as some time must be left for the GPS/IMU program to perform its actions.

3.5. Path Generating Algorithm

3.5.1. Description of the Path Generating Algorithm

The Path Generating Algorithm (PGA) is the heart of the HUD system. This algorithm is responsible for calculating the path that the paratrooper should follow, with the aim of allowing the paratrooper to jump as far away from the LZ as possible. This is desirable, because increasing the stand-off distance lessens the chance that the enemy will be able to determine the target of the paratroopers. Aircraft are noticeable; a small parachute, moving silently through the air at night, is not.

The algorithm takes as inputs the wind information (a list of at least two and up to an arbitrary number of altitudes, each with an associated speed and direction), the parachute performance characteristics (the rate of descent and forward speed), and the position of the LZ (a latitude, longitude, and altitude). The output consists of a set of waypoints, each defined by a three-dimensional coordinate, and a heading that the paratrooper should fly towards for each waypoint. The algorithm stores its output in a structure for future use; this structure remains static once the path is generated. The inputs and outputs are illustrated in Figure 19.

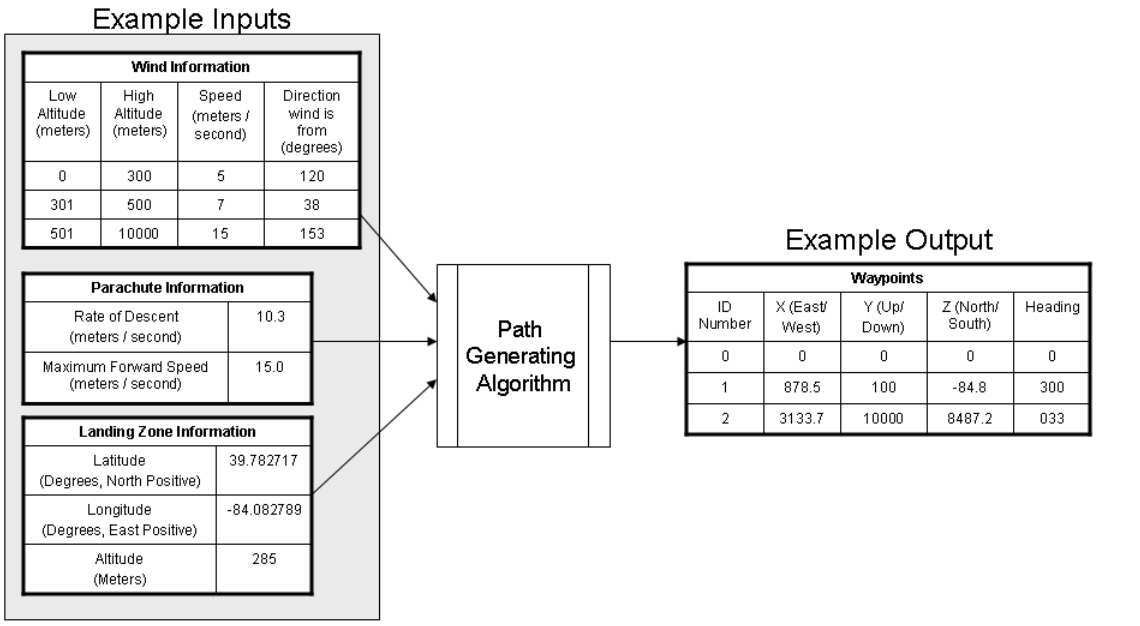


Figure 19: Example Inputs And Outputs To The Path Generating Algorithm

The waypoints use X, Y, and Z Cartesian coordinates instead of latitude, longitude, and altitude, because the distances can be more easily calculated, and because a Cartesian system works well with Allegro's coordinate system. The X axis corresponds to east and west distances; east is positive. The Y axis corresponds to up and down distances; down is positive. Although unintuitive, the Y axis is flipped within Allegro's coordinate system (where the top of the display is at the 0 Y coordinate, and the bottom of the screen is a positive integer). The Z axis corresponds to north and south distances; north is positive. Each unit in the XYZ coordinate system is equivalent to 1 meter in the real world. The landing zone is defined to be located at (0, 0, 0). All other waypoints are defined relative to the landing zone.

The pseudocode for the algorithm is illustrated in Figure 20. The size of the problem is defined by the number of levels of wind speed. The algorithm begins at the ground, and works its way up to the highest altitude for which wind data is present.

```
Set LZ at 0, 0, 0
Determine wind speed and direction at LZ's altitude
Generate waypoint so that paratrooper lands facing the wind
For every wind level
        For headings 0 – 359
                Determine how far paratrooper can travel
                Measure distance from the LZ
                Record maximum distance from the LZ
        Set new waypoint at maximum distance from LZ
        Increase altitude to next wind level
```

Figure 20: Pseudocode For The PGA

The PGA begins by examining the wind around the LZ. Parachutists, like aircraft, like to land into the wind, as this minimizes their groundspeed and makes the actual landing softer. The algorithm selects a heading 180 degrees from where the wind is blowing, so that the paratrooper is facing directly into the wind. The turn point is set at 100 meters above the LZ's altitude. The algorithm uses the paratrooper's rate of descent and forward speed to determine how far he can travel during the last 100 meters of descent, and generates an appropriate waypoint. The waypoint is stored in a waypoint structure for future use.

Once the turn waypoint has been set, the algorithm begins generating the rest of the path. The algorithm assumes that the 'current altitude' is the same as the last waypoint generated. The algorithm then gathers information about the wind. The wind is treated as being 'sliced' into several layers. Each layer has a low altitude (the lowest altitude for which the wind information is valid) and a high altitude (the highest altitude

for which the wind information is valid), as well as a speed and direction that the wind is blowing from. The algorithm determines which layer of wind affects its current altitude, as well as the differences between the current altitude and the high altitude of the layer. By using the parachute's rate of descent, the algorithm can determine how long it will take the paratrooper to travel from the high altitude of the wind layer to the last generated waypoint.

Once the time it will take the paratrooper to descend is calculated, the algorithm determines how far the paratrooper can travel in that time. This is done by assuming that the paratrooper follows a constant heading from one waypoint to the next. The algorithm iterates through 360 degrees of heading, calculating how far the paratrooper could travel at that heading, given the wind conditions and the capabilities of his parachute, within the previous-determined timeframe. The program chooses the heading that places the paratrooper furthest away from the LZ. This point is selected as the next waypoint. The algorithm then repeats the process for every wind layer. By always choosing the point furthest away from the LZ, the algorithm attempts to maximize the distance between the LZ and the RP.

The end result is that the last waypoint generated is the point at which the paratrooper should begin navigation. The paratrooper will have to exit the plane above the waypoint, but paratroopers have access to existing tables that determine their forward throw once they leave the aircraft that will enable them to accurately exit the plane, deploy their parachute, and begin navigating in the vicinity of the last waypoint generated.

The path generating process can be visualized as drawing a 360-sided polygon around a waypoint, and then finding which vertex of the polygon is furthest away from the LZ. The process is illustrated in Figure 21 and Figure 22.

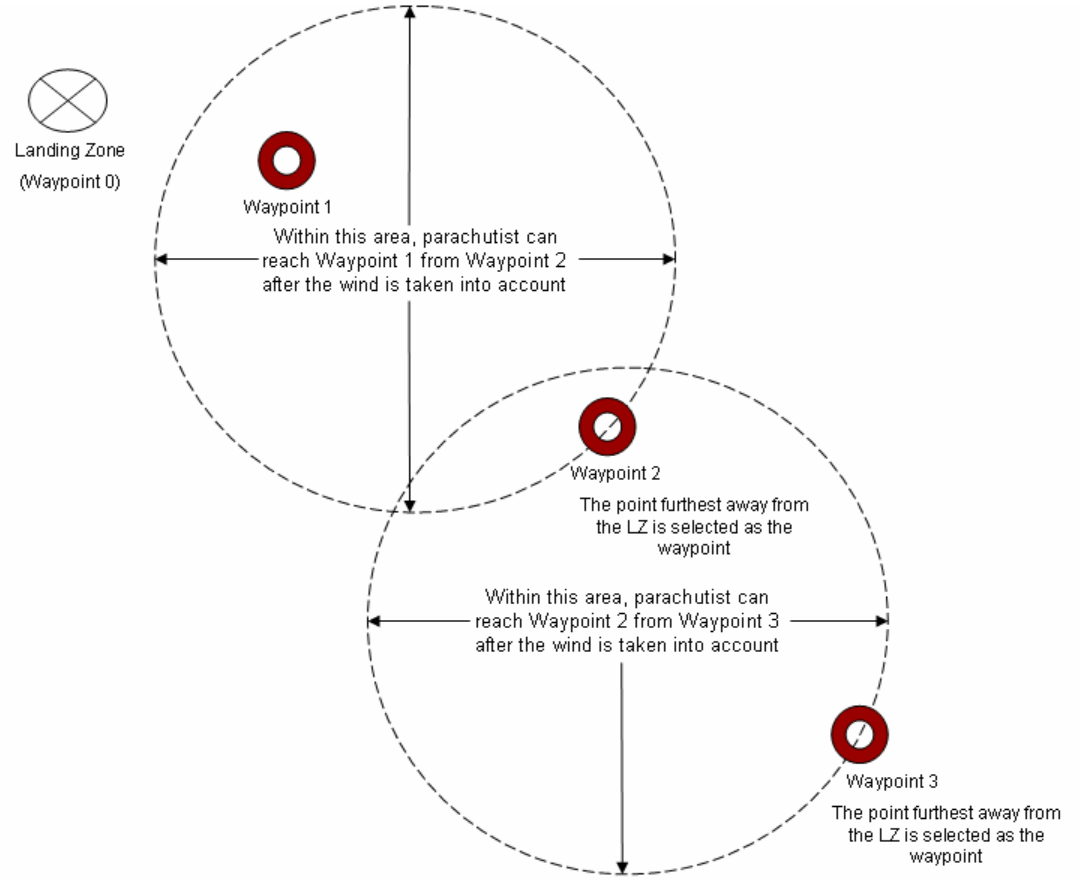


Figure 21: Top-Down View Of The PGA's Progression

Viewed from above (Figure 21), the PGA's progression appears to be a number of overlapping circles (or 360-sided polygons). The point on the circle furthest from the LZ is selected as the next waypoint. By always picking the furthest point, the algorithm is able to generate the longest possible path for the paratrooper to follow, given the assumptions stated in Section 1.5.

When viewed from the side as in Figure 22, the notion of wind layers becomes clear. Each waypoint marks the boundary between two wind layers. The final waypoint is located at the top of the last wind layer; in Figure 22, Waypoint 3 is where the paratrooper is expected to begin navigation. Since it is assumed that the wind speed and direction within a wind level remains constant, generating the waypoints is simplified. Testing must be conducted to ensure this assumption is valid.

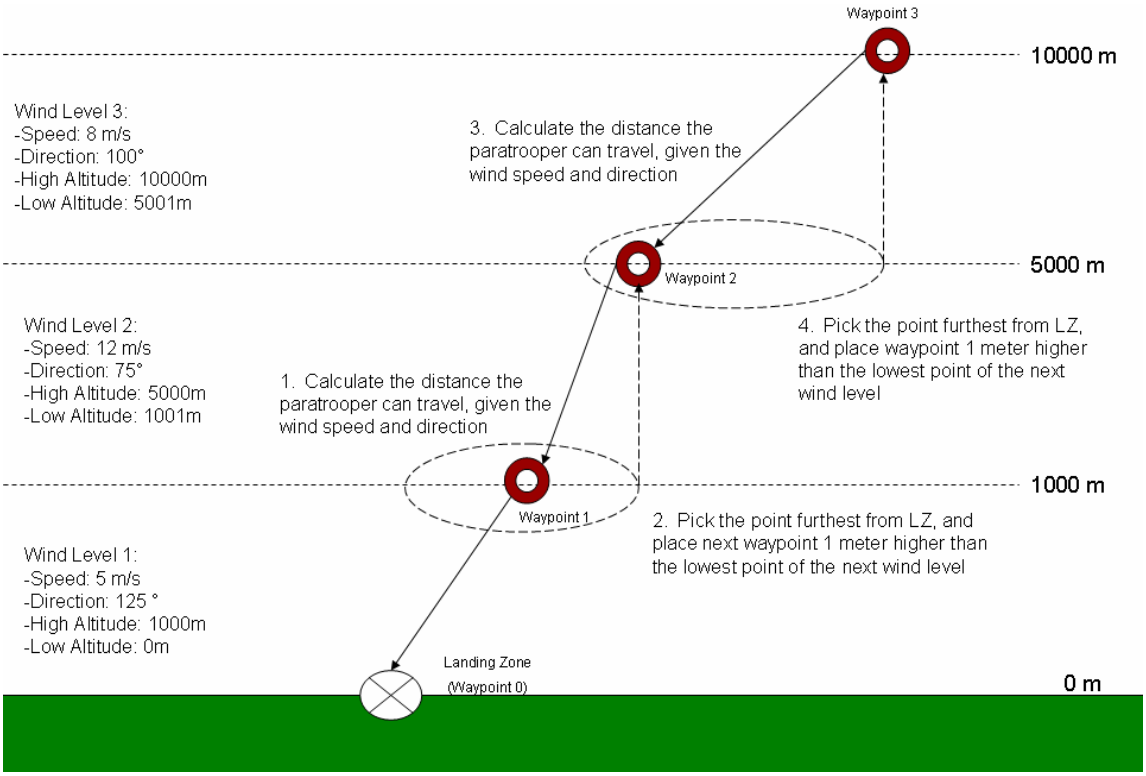


Figure 22: Side View Of The PGA's Progression

3.5.2. Analysis of the Performance of the Path Generating Algorithm

As the HUD software will be running on a relatively low-powered computing platform, an analysis of its performance is appropriate. A perfect algorithm that takes an unreasonably long time to compute a path is useless when a mission must be executed as quickly as possible.

The inputs to the algorithm are a set of wind data, parachute information, and landing zone information. Both the parachute and LZ inputs are of a fixed size. The wind data can vary in size, from only one layer of wind data, up to any arbitrary number of levels. As the number of layers of wind increases, the execution time of the PGA will increase.

A simplified version of the PGA's pseudocode is presented in Figure 23. As can be seen, there is a nested for loop. This is an indicator that the algorithm could scale badly. However, the inner for loop runs in O(1) time, as it always calculates the distances for 360 different headings.

```
Generate Waypoint 1

For every wind level

        For every heading

                Calculate distance

                Set Waypoint at furthest distance
```

Figure 23: Simplified Pseudocode of the PGA

Thus, the algorithm can be treated as if it only has one for loop within it. The input to the outside for loop is the wind data. As the outside for loop must run once for every wind level, the algorithm runs in O(n) time – that is, as the number of inputs increases, the algorithm's running time increases linearly. Given that there is a limit to the granularity with which wind speed and direction at different altitudes can be measured (usually every 150 meters) and a limit to the maximum altitude that HAHO jumps can be conducted at (less than 8,000 meters), the algorithm should scale well for all foreseeable situations.

3.5.3. Weaknesses of the Path Generating Algorithm

The PGA has the singular goal of generating a set of waypoints that guarantees a paratrooper that he will reach the LZ, as long as he visits the waypoints. If the PGA generates a path that makes it physically impossible for the paratrooper to follow, the algorithm has failed. There are three areas that could potentially lead to the algorithm failing: the parachutist not following the guidance data, inaccurate wind data, and inaccurate modeling of the parachute's performance. The parachutists' navigation is outside of the control of the software – the final path traveled is ultimately up to the human guiding the system. However, inaccurate wind data and inaccurate modeling of the parachute's performance can be addressed by the system.

Inaccurate wind data is a major problem, as paratroopers are at the mercy of the wind (compared to other forms of aerial transportation). While ram-air parachutes do have the capability to navigate, gravity means that they can only travel so far before they reach the ground. Parachutists do not have any form of propulsion that can push them against the wind. Therefore, if the wind data supplied to the PGA is inaccurate, the path generated may not be feasible.

The wind data is collected by the aircraft carrying the paratroopers. The aircraft flies at a low altitude, searching for the target. Once the target is located, its coordinates are entered into the aircraft computer, and the aircraft climbs to a safe jumping altitude. During the climb, wind data is gathered [STS03a]. This wind data can be out of date by the time the aircraft reaches a suitable altitude, or the aircraft could have been flying through a transient gust of wind that could throw off the wind speed readings.

One solution would be to implement a sliding scale that could be adjusted by the user, which reflects the confidence level in the wind data. If the wind data is fresh and predictable, then a high confidence level can be placed in the wind data. If the winds seem to be highly variable, a low confidence level can be placed in the wind data. Depending on the confidence level placed in the wind data, the program can generate a more conservative path. A more conservative path means that parachute is thought to descend more quickly than it actually does, so that the algorithm underestimates the performance of the parachute. This will give the paratrooper some margin for error.

Errors in modeling the parachute's performance can also lead to errors in generating an optimal path. A parachute's age directly impacts its rate of descent, for instance. A new parachute's cloth is more tightly woven than an older parachute's cloth, which leads to increased air resistance. Thus, a new parachute descends slower than an old parachute. Also, the proper parachute performance parameters must be submitted to the PGA in order for a proper path to be generated. If paratroopers receive a new model of parachute, this information will need to be obtained and recorded so that it can be readily available when the HUD system is to be used.

The algorithm assumes that the parachute behaves as if it is attached to the wind. For instance, if the wind is blowing from the south at 10 m/s, the parachute will travel 10 meters north in one second. This assumption is a reasonable one [BAL03]. Algorithms to more accurately model the motion of a parachute have been developed. However, most algorithms model the motion of simple round parachutes, which have no ability to guide themselves. The ram-air parachutes used by paratroopers do have this ability.

Other algorithms are simply too complex. A model of a ram-air parachute aerodynamics was discovered; the model is a finite-element analysis of the airflow around the airfoil and designed to be run on a supercomputer [MIT97]. This is obviously not feasible for the low-powered computer used in this thesis. Empirical testing will be necessary to determine what an appropriate trade-off between realism and computational efficiency will be.

Another improvement would be to also implement a turn-rate limiter. Currently, there is no provision to ensure turns in the path, located at each waypoint, are within the parachute's turn rate. Although there is no real penalty for flying out of the tunnel, it may be psychologically more acceptable to design a tunnel with no sharp turns that would be difficult for the paratrooper to follow.

3.6. Generating the Virtual World

The virtual world consists of two primary structures – waypoints and the path between waypoints. Waypoints are represented as spheres approximately three meters in diameter. The path between waypoints is represented as a series of boxes, each ten meters per side, forming a virtual tunnel to fly through. Although the two structures are similar in implementation, there are some differences that are worth investigating.

Every object in the world is formed out of a number of vertices. Allegro defines a vertex's properties, which include a set of X, Y, and Z coordinates. By creating a number of vertices and defining their locations, one can create a many-sided polygon. Many polygons could be joined together in order to form very complex shapes; a group of vertices to form a shape is called a model. However, due to the requirements of this

thesis, only simple shapes are used. This is necessary both to cut down on clutter in the HUD and to improve the speed at which a frame can be generated.

Allegro provides functions to manipulate the vertices; these functions include rotations, translations, and scaling. By applying these functions, a single set of vertices defining a shape can be used over and over again, defining many copies of a single shape. Only a single model has to be created. Once the model is created, it can be rotated and translated to the proper virtual-space position. Another model can be created, and the second model can then be rotated and translated to another virtual-space position.

3.6.1. Waypoints

Waypoints are composed of two vertices and appear to the user as a solid sphere or circle. These vertices are spaced 3 meters apart from each other, and used to determine the diameter of the circle. Each waypoint is colored green if the user's altitude is greater than the waypoint's, and red if the user's altitude if less than the waypoint's. The current HUD system is monochrome; STS anticipates fielding a color HUD soon, which will be able to take advantage of the colored waypoints.

At far distances, a waypoint's size on the screen may be less than a pixel. Because waypoints are important, all waypoints will be drawn so that they are at least one pixel tall, regardless of their correct, sub-pixel, size. By drawing all of the waypoints, the user's path can always be seen, no matter the distance.

3.6.2. The Path

The path is composed of a series of boxes, each placed 100 meters apart from each other. Each box is 10 meters per side, providing a large enough space for a

paratrooper to comfortably fly through, but still small enough to indicate precisely where to fly. Each box is modeled as four points, each one 5 meters above or below the waypoint, and 5 meters east or west of the waypoint. Although all are initially located at the origin when created, they are individually rotated and translated so that they are aligned with the straight line path between waypoints.

The boxes are designed so that although they are rotated around the Y axis to align with the straight line path, they are not rotated around the X axis – that is, they are all perpendicular to the ground. Initial testing during development indicated that having boxes perpendicular to the ground was more readily understandable to the user than having boxes perpendicular to the path of travel.

3.6.3. Weaknesses of the Virtual World's Implementation

The primary concern with the virtual world's implementation is performance. As the distance to be traveled increases, so does the number of vertices that must be generated. The rotation and translation of a vertex to its appropriate spot is somewhat processor-intensive. To increase performance, the number of waypoints and boxes drawn should be kept to a minimum. However, there is a tradeoff in presenting data to the user. If only a few boxes are drawn, then the user is not receiving enough useful information.

One method to help improve the performance of the system would be to omit the drawing of boxes that are sufficiently far away from the user. This would eliminate the computing time necessary to translate a box from the virtual world to screen space. A complex, fully-drawn path may fill up the screen with many small boxes that could confuse the user.

Another method that may not improve performance, but may cut down on clutter, is to shade the boxes and waypoints to give the illusion of distance. A box that is farther away from the user could be colored a dark gray, while a box close to the user could be a bright white color. A downside to this approach is the limited color palette of the HMD display. It may not be able to accurately differentiate between subtle gray shadings. Future versions of the HMD display may solve this problem.

3.7. Design of the HUD

The user's view of the system is the HUD. The HUD is responsible for conveying all of the information that the system generates to the user. The information must be presented clearly and completely. The HUD must also be legible despite the limited screen size and resolution. The NVGs that this thesis developed the HUD for are AN/PVS-21 goggles, produced by Specialized Technical Services of Beavercreek, Ohio. These goggles utilize a unique beamcombiner which combines the enhanced image from the night vision tubes with an external video source, and displays the combined image to the user. The display is monochrome and runs at VGA (640 x 480 pixels) resolution. The display subtends 13 degrees of the user's field of view, which is equivalent to viewing a 17" computer monitor from 5 feet away. Figure 24 shows a simulated view of the HUD through the NVGS.

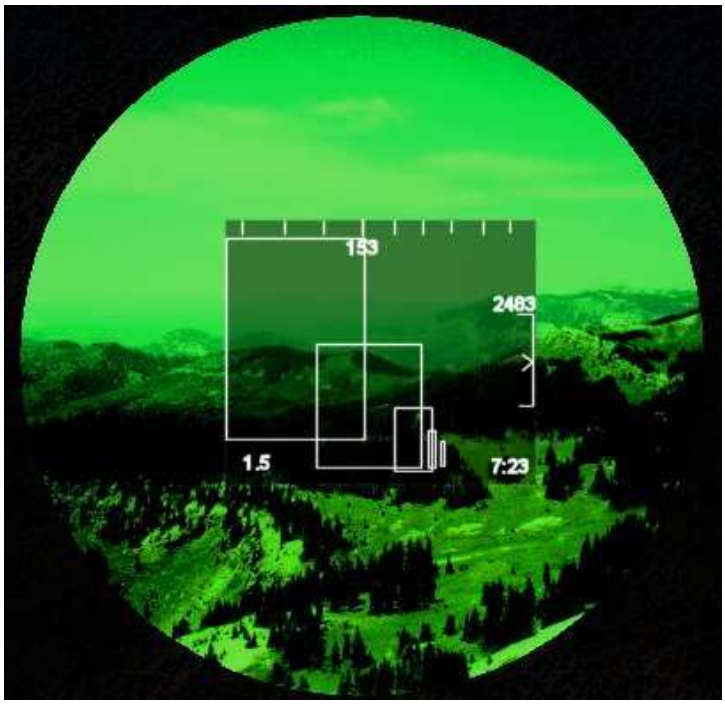


Figure 24: Simulated View Of The HUD Through The NVGs

3.7.1. Elements of the HUD

During the parachute jump, the paratrooper must know several key pieces of information in order to successfully complete the jump. This information includes the paratrooper's current heading, his altitude, his groundspeed, his time to go, his distance from the LZ, and the path he should follow [BAL03]. The HUD is designed to convey this information in an easily understandable format, as seen in Figure 25.

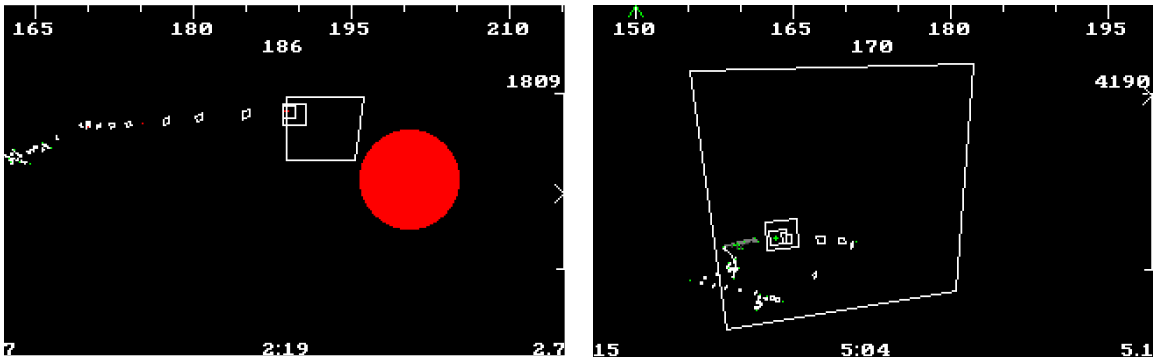


Figure 25: Simulated HUD Displays

Across the top of the display is a heading scale, which gives the paratrooper a quick reference to his heading. A green caret indicates the heading that the paratrooper should fly in order to reach the next waypoint. His exact heading is displayed in the center top of the display. The paratrooper's current altitude (in meters) is displayed on the right side of the HUD, above an altimeter. The top line of the altimeter represents the RP's altitude, and the bottom line represents the LZ's altitude. The altimeter's caret points to the paratrooper's relative altitude above the LZ.

Along the bottom of the display are three numbers. The leftmost number is the paratrooper's ground speed (in meters per second) towards the LZ, or his "Velocity Made Good." The center number is the estimated time the paratrooper has until he reaches the ground. The rightmost number is the paratrooper's horizontal distance (in kilometers) from his current position to the LZ.

The center of the screen is devoted to displaying the path that the paratrooper should follow. The path is made up of the spheres and rectangles discussed in section 3.6.

In order to ensure that all screen elements are legible, care must be taken in order to prevent one screen element from overwriting another one and obscuring that information. For instance, there may be situations where a long line of boxes forms a thick, long line across the screen. It would be annoying at best if this white line of boxes overlaid the heading or altitude information. In order to prevent this, a technique called 'haloing' is used [PAR03]. All screen elements are outlined in black, creating a halo that prevents elements from overlaying each other. Also, the path information is drawn first,

with the other fixed HUD elements being drawn over the path. This prevents the path from obscuring the fixed data.

3.8. Summary

This chapter discussed the details of the software portion of the NVG/HUD program. The Allegro graphics library and its particular suitability for this project were discussed. Next, various aspects of the HUD software were discussed. The Path Generating Algorithm is the heart of the HUD system, as it is responsible for generating all of the waypoints that make up the path that the paratrooper is expected to follow. The algorithm itself, as well as its potential weaknesses, was discussed. The structures making up the virtual world were also described. Finally, the HUD itself is discussed, including the layout of the HUD, and the hardware used to display the HUD to the user.

4. Methodology

4.1. Chapter Overview

This chapter discusses the techniques used to evaluate the graphical HUD developed by this research. The HUD was evaluated on its ability to guide a user along a path efficiently and accurately. The HUD was tested against a baseline GPS navigational reference.

First, the goals of the HUD evaluation were defined. The boundaries of the system were defined, and the component under test was defined. Parameters of the testing, and factors that may affect the outcome of the test, were identified and their levels enumerated. Finally, a methodology to test the HUD is proposed.

4.1.1. Goals and Hypothesis

One of the objectives of this thesis is to compare the effectiveness of various layouts of navigational information on a HUD. Figure 26 and Figure 27 depict the different visualization methods under test.

Figure 26: View Through The NVGs Without Any Navigational Aids. The User Will Be Equipped With A GPS Displaying The Latitude, Longitude, And Direction Of Travel.

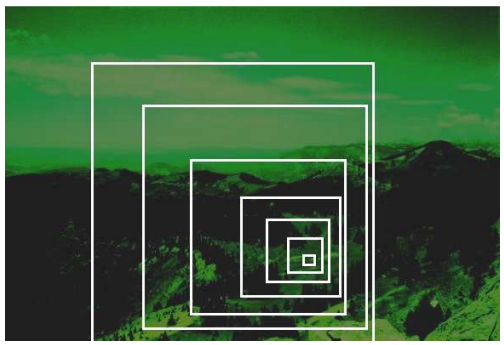


Figure 27: View Through The NVGs Using A Tunnel In The Sky System. By Flying Through The Tunnel, Which Depicts The Generated Path To LZ, The Paratrooper Will Be Guided To The LZ. If The Tunnel Is Not Visible, An Arrow Will Appear At The Side Of The Screen.

Trying to navigate to a LZ consisting of a small circle with a 30ft radius, without a navigational aid, at night is extremely difficult, yet Special Forces troops manage to accomplish this difficult feat. By adding basic navigational information (as seen in Figure 1 on page 4) troops can better gauge their distance to the LZ as well as their relative position. Unfortunately, most of the information presented is textual and hard to read and process in high-stress situations. In addition, the exact location of the LZ and the trooper's position relative to it is not explicitly presented. Troops must cross-reference heading, altitude, and velocity readings with a chart pinned to their sleeves to make sure that they are on course. This information is useful, but cumbersome to utilize. It is hypothesized that a graphical display guiding paratroopers to the LZ would result in higher accuracy, with less effort.

Display area is at a premium in NVGs. NVGs allow only a 40 degree field of view. Current HUDs only occupy about 13 degrees in the field of view. This is equivalent to viewing a 17" computer monitor from 5 feet away. Therefore, graphics cannot be too detailed or they will be completely unusable. Figure 27 is an example of a tunnel in the sky system that takes advantage of three-dimensional graphics to concisely present information. A virtual 'tunnel' of rectangles is displayed, showing the optimal path from the release point (where the paratrooper jumped out of the plane) to the LZ. By flying through the tunnel, the paratrooper is guaranteed to land at the appropriate target. The trade off made by displaying more information is in graphical complexity and simplicity. It is possible that the large number of rectangles necessary to map out a path will clutter the tiny display screen excessively.

4.1.2. Approach

The research goal is to compare several different methods for conveying navigational information to a paratrooper via a HUD. To identify the superior system, direct comparisons between the various methods was accomplished. Due to the expense and danger involved in conducting actual parachute jumps, ground testing was conducted (a justification for this choice is provided in Section 4.3, on page 66). Empirical measurements of tests conducted on the ground by subjects using a HUD displaying navigational information were collected.

A number of paths, each made up of several waypoints, were established. Each path was followed by using navigation information gained from one of several sources. The first source was a GPS system displaying a pointer to the next waypoint, to establish the baseline performance. The baseline performance was then compared against a graphical HUD (Figure 27).

4.2. System Description

The components that made up the System Under Test (SUT) include:

- The GPS system, consisting of a Garmin RINO 120 receiver collecting navigational information, including latitude, longitude, and altitude, from the GPS satellite constellation.
- The test subject, as this individual was responsible for acting upon the supplied navigational cues.
- A Toshiba A15-S129 laptop, which collected information from the GPS as well as performs calculations for the HUD software system. The laptop was equipped

with a Celeron CPU running at 2.4 gigahertz, with 512 megabytes of RAM. The system ran Red Hat Fedora Linux Core 2 on a VMWare Virtual Machine.

- The HMD hardware, specifically a STS M-HUD Model 2750.
- The Guidance System. The HUD software ran on the laptop and used data collected from the GPS to determine where the user was and where he was looking. The HUD software generated and sent the appropriate graphics to the NVG system. Alternatively, the user observed the GPS's directional arrow, omitting the laptop completely.

The component under test (CUT) is the type of HUD graphical display. Two different types are identified (one baseline and one alternative).

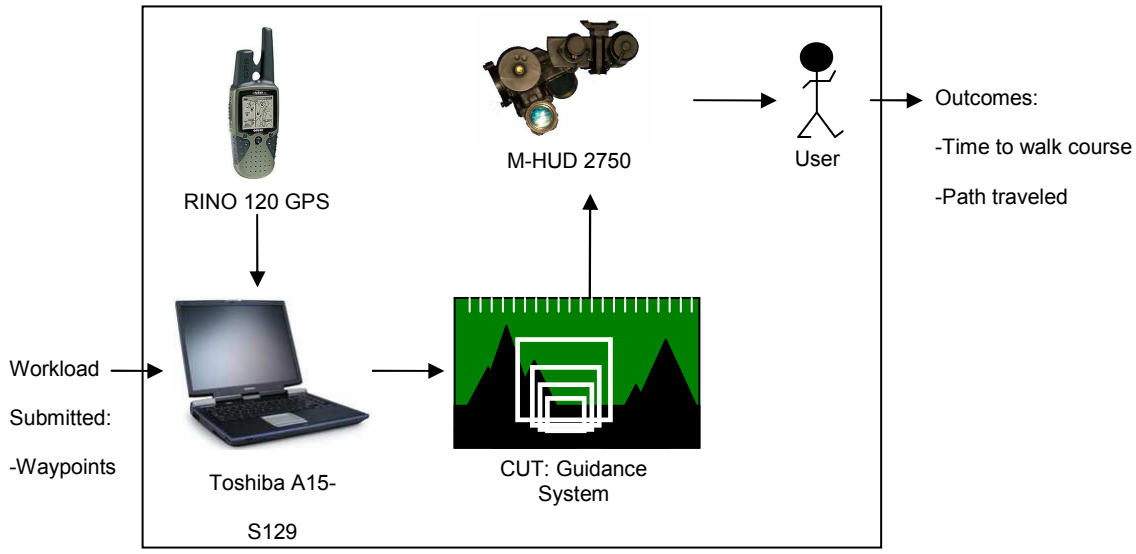


Figure 28: The System Under Test

4.3. Evaluation Technique

Evaluating this system requires taking measurements of the actual system. There are no analytic or mathematical models that can adequately capture the way a paratrooper

reads a HUD and reacts to the guidance provided. Therefore, analytic models are inappropriate for this type of testing. Simulations present another, better, choice for testing this system. However, there are several reasons why simulations are also inappropriate.

Simulations can achieve varying levels of realism. Basic simulations of a HUD have been conducted on personal PC's, for instance. These simulators present users with a simulated view of the HUD, and allow the user to steer by using the arrow keys on the keyboard [BAL03]. Obviously, this is a very rudimentary simulation system, and while it can be useful for preliminary results, further testing is desirable for a system that will eventually be used in life-threatening situations.

Ideally, the system should be tested with a parachutist actually jumping out of airplane, in order to accurately measure how effective the path-generating algorithm is. Parachute jumps are, by their very nature, dangerous and expensive. Also, full testing would require the participation of several trained parachutists, who are in short supply. Because of these reasons, conducting testing on the HUD via parachute jumps is considered impractical for this research.

Fairly complex parachute simulators exist. These may involve a full mock parachute harness, and are often used to give military aircrews parachute refresher training. There are many desirable characteristics about these simulators, including the ability to extensively log data, but there are also a few disadvantages that make simulators unsuitable for testing this system. These simulators require the use of a set of virtual reality goggles, which precludes wearing a set of NVGs. Further, while these

simulations can imitate many different weather conditions, none have been programmed to simulate the wearing of NVGs, and none have been programmed to emulate the various HUD systems under test. While this could be done, this approach was not pursued for this particular project.

A safer and more practical testing system is to test the system on the ground, modifying the software so that instead of computing a path, it receives a pre-computed path to display. This testing method has several advantages. First, it is both much safer and much less expensive than actual parachute jumps, test subjects (walking people) are readily available, and test conditions are more easily duplicated. Wind conditions change constantly; two parachutists jumping two hours apart may encounter two totally separate wind conditions. On the ground, a single path can be walked multiple times by multiple test subjects, and conditions will not change greatly.

There are two major disadvantages to conducting testing on the ground. First is the loss of realism. Walking is a two-dimensional navigation problem; parachuting is a three-dimensional navigation problem. Once a parachutist descends too far below the optimal path, it may very well be impossible for him to regain that path. The parachutist is also at the mercy of the winds, and must account for the wind's effect on his path when trying to navigate. A person walking on the ground is under no such restrictions. The second problem is the inability to test the path-finding algorithm. The path-finding algorithm is critically important when generating a path for a parachutist. The function of the algorithm must be verified somehow, as it is useless if it gives the paratrooper a path that he physically cannot follow.

Since the HUD system has already been implemented in a stand-alone computer system and is available, it makes the most sense to test the actual system. The data is also easily verifiable, since real-world measurements were taken at the time of the experiments.

4.4. Experimental Design

The basic experimental procedure was to take the test subject to a starting location, and equip him with a guidance system. The navigational equipment was pre-programmed with a set of waypoints, forming a path, which the subject arrived at in order before arriving at a destination.

All of the walks took place on a night without inclement weather. Subjects wore clothing appropriate to the weather, including footwear suitable for hiking. Before the test began, three paths, each consisting of a number of waypoints, was generated by the test monitor. Each path was on a stable, safe location on Wright-Patterson AFB. Care was taken to place waypoints at locations that are not hazardous (such as the firing range or base entry points) or off-limits to walkers. Arriving at the waypoints did not require any strenuous activity, such as climbing. However, in some of the paths, obstacles such as trees and buildings were present. Each path was less than 1 mile long and took less than 20 minutes to complete.

The specific path that the user took was selected before the test began. The path was selected by the test monitor and loaded into the HUD software and GPS. The monitor then checked all of the HMD equipment to ensure that was working properly, including receiving readings from the GPS system. The subject then donned the test

equipment, and moved to the start waypoint. The test monitor started the stopwatch and instructed the test subject to begin navigation. This was the start of the test. The subject followed the navigational cues presented on the HMD or GPS until he arrived at the last waypoint. The monitor then stopped the stopwatch and recorded the elapsed time.

The test monitor accompanied the test subject in order to ensure that they completed the entire course. The monitor warned the test subject of any safety hazards the test subject was approaching (such as a hole in the ground or a street), but did not offer any guidance clues. The monitor did not provide any of the test subjects the exact location of the waypoints, or indicate to the test subject if they had come close or not to the waypoint.

Testing began by using the GPS as the guidance system. Each user walked Path 1 using the GPS. Their times were recorded by the test monitor. Each user then walked Path 2, and the times were again recorded. Each user then walked Path 3. After all of the paths had been navigated, each user was equipped with the HUD and walked Path 1. Times were recorded. Path 2 and Path 3 were again walked in the same manner.

Although it could be argued that allowing each user to walk the path twice could result in biased test results due to increased familiarity of the path, measures were taken to prevent this. Tests were conducted at night, to minimize the user's ability to see surrounding terrain and landmarks. Each walk by a user was conducted at separate times, so that the layout of a particular path was not fresh in a user's mind. Users were not told which path they were following when beginning the test. Users were not told the location

of waypoints except by the navigational references. Because of these precautions, it is believed that the data collected is unbiased.

4.5. System Services

The NVG/HUD system's primary service is a set of directions along a path. Depending on the exact HUD used, the format of the directions varies. For instance, with no HUD, the set of directions came only from the GPS's pointer. At the other end of the spectrum, the Tunnel in the Sky HUD provides a very explicit set of directions along the path.

In the test, the quality of the services provided was measured by the time taken to navigate the entire course. Lower times are judged to be better than higher times. If the directions provided by the CUT are easily understandable, the user will be able to navigate along the course rapidly, without having to take time to decipher the navigational cues. If the directions are not easily understandable, the user will be forced to stop and carefully examine the directions provided to him, slowing him down. Lower times therefore indicate that the guidance system in use is efficient.

4.6. Workload

The workload submitted to the system consisted of a set of waypoints, each consisting of a latitude, longitude, and altitude. The system then drew a path between each waypoint. The first waypoint submitted to the system was designated as the starting waypoint, and the last waypoint submitted to the system was designated as the ending waypoint. The other waypoints were used to define the path. The path was made up of a number of boxes between each waypoint, forming a tunnel for the user to travel through.

4.7. Performance Metrics

The primary service, a set of directions from the start waypoint to the end waypoint, was evaluated based on how efficiently it is able to guide the user along the course. Therefore, the primary performance metric was the time taken by the user to traverse the course. It is assumed that the more intuitive that a display is, the quicker the user will be able to follow the navigational cues to each waypoint.

4.8. Test Parameters

The system has several parameters that affect its performance. These include:

- Environmental conditions. This includes the air temperature, precipitation levels, and the terrain being navigated across.
- The user. Different users have different walking speeds, and different levels of stamina. Users also differed in their ability to understand navigational cues presented to them.
- GPS accuracy.
- The Guidance System. Each method of displaying information has strengths and weaknesses, with the primary tradeoff being between display complexity and information conveyed.

The following are parameters of the workload submitted to the system:

- Waypoint coordinates. This information, consisting of a latitude, longitude, and altitude, changed with every test.

4.8.1. Test Factors

The following parameters were factors in the ground testing of the SUT.

- Guidance system. This system has two levels – the GPS pointer or the graphical HUD.
- Time to traverse the course. Every user will complete a course in a different amount of time. Lower times indicate that the navigational reference used performed better.

Some of the parameters were not chosen to be varied during testing. For instance, the ability of the user may very well be a "factor" in the performance of the CUT. However, for the purpose of testing, it was assumed that all members, since they were in the Air Force, were in good physical shape and were capable of walking at a normal speed for a short period of time.

4.9. Suggestions for Future Testing

Ground testing was selected as a safer, less expensive way of testing the system. However, ground testing bears little relation to actual parachuting. Testing of the system with parachute jumps is necessary to ensure that it performs as intended.

The system components necessary for parachute testing would be similar to the components of the system tested on the ground with the exception of the MT-9 IMU. The MT-9 IMU would be a vital component in parachute testing, as the paratrooper's head orientation would be impossible to calculate from GPS velocity information alone. Implementing head tracking is necessary before any parachute testing could occur.

In parachute testing, the primary service provided by the system would be a path generated by the PGA from a RP to a LZ. By using the PGA-generated path as the system service, the performance of the PGA could be evaluated.

The workload submitted to the system is a request for a set of directions from a release point (RP) to a landing zone (LZ). These requests are submitted to the system by defining a three dimensional point for the LZ, consisting of latitude, longitude, and altitude coordinates. In order to compute the path, the system also requires that the user submits wind data, consisting of wind speed and direction at different altitude levels, as well as performance parameters of the parachute, such as the rate of descent and the maximum forward velocity. Once this information is entered, the system generates a path, placing the release point as far away as possible from the LZ, given the performance characteristics of the parachute.

The possible outcomes of the primary service provided by the NVG/HUD system are a successful landing in the LZ, or an unsuccessful landing away from the LZ. However, there are shades of 'successful.' If a paratrooper is aiming for a 30ft circular LZ in the middle of a field and he lands within it, then he has made a successful landing. Missing the LZ by two feet does not make the landing unsuccessful in this case. On the other hand, the 30ft LZ may be the roof of a tall building. Landing two feet outside of the LZ will be unacceptable, since the paratrooper can be seriously injured or killed if he falls to the ground from the roof. Therefore, the outcome can fall into three levels of success – Total Success, where the paratrooper lands within the specified LZ, Acceptable Success, where the paratrooper lands outside of the specified LZ, but within a range such that there is no mission impact, and Failure, where the paratrooper lands so far away from the LZ that he could be injured, or the mission is severely impacted.

The primary service, the guidance information from a RP to a LZ, should be evaluated based on how close to the LZ it can guide the user. Therefore, the performance metric should be the distance from the LZ that the paratrooper lands, measured in meters, from the center of the LZ. This particular metric is chosen since the purpose of the system is to get the paratrooper as close as possible to the center of the LZ. Comparing the measured distance from the LZ with the acceptable distance away from the LZ (as specified in the factors of the workload) makes it possible to determine if the user achieved a Totally Successful outcome, Acceptably Successful outcome, or a Failed outcome.

Wind speed can have a major impact on the performance of the paratrooper. While the wind is outside of the control of the testers, the wind conditions should still be recorded so that is impact on the test can be analyzed. Pertinent wind conditions include wind speed, direction, and the altitude for which the wind information is valid.

Testing of the system should encompass as many jumps as possible, given budget and personnel constraints. However, full factorial testing of the system will rapidly increase the number of jumps. Pilot testing should be used to identify factors that have minimal impact on the performance of the system. This will reduce the number of jumps necessary to test the system.

4.10. Summary

A NVG HUD system has been developed, and it was necessary to test the various methods of displaying navigational information to determine which system is the best. "Best" is defined as the HUD that can guide a user along a path the quickest. It was

hypothesized that a graphical HUD will perform better than other systems. To prove this hypothesis, a number of tests were conducted, each testing the various HUD layouts under different test conditions.

The entire system consists of the M-HUD, HUD software, the laptop computer system, the GPS system, the user, and the environment; the HUD software will be the CUT. These components all work together to shape the way that the system provides its service: a set of directions along a path.

Although there are many parameters that affect the system, including the HUD, the user, the environment, the GPS system, etc, the factors tested were the guidance system and the time to navigate the course. The system was measured by collecting data on how quickly the user is able to navigate the course. Ground testing was utilized due to the expense and danger involved in testing the system by parachute jumps.

Suggestions are provided on how to test the system more realistically by conducting parachute jumps.

5. Test Results

5.1. Introduction

Testing of the HUD presented several challenges that had to be overcome. However, testing was conducted and data gathered. It was found that one of the performance metrics for which data was gathered did not yield any useful results. In this chapter, an analysis of the data gathered is presented, along with explanations for the results.

5.2. Test Design

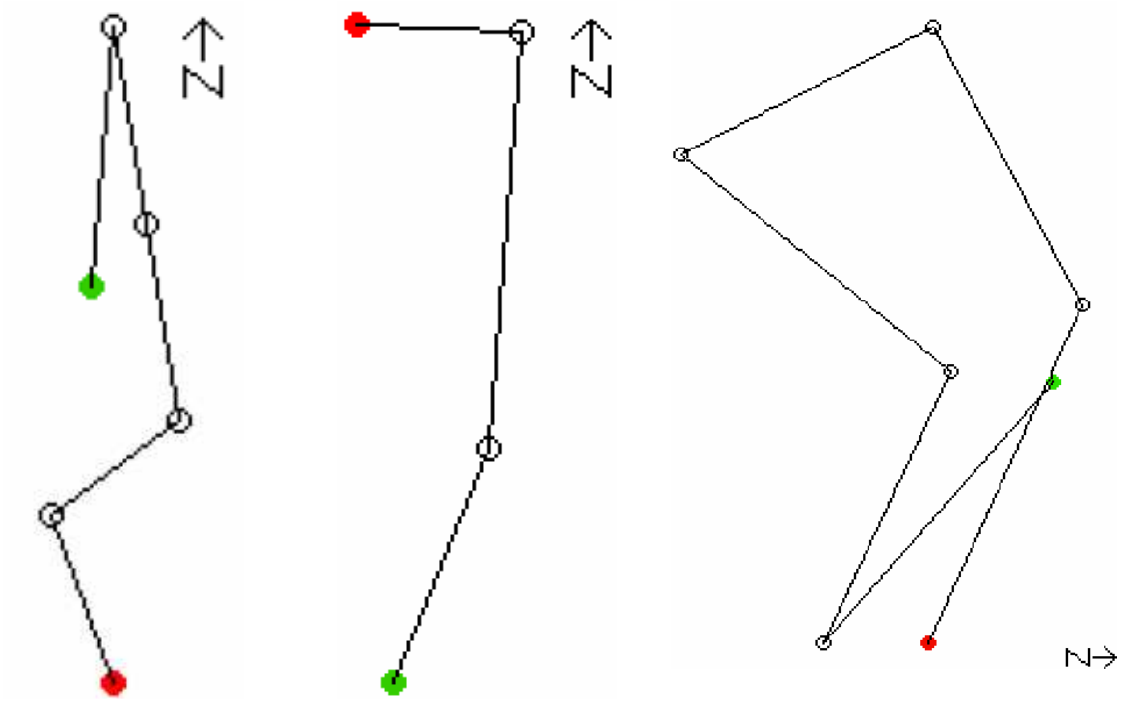
The HUD software under test differed in some aspects from its initial specifications. Since the IMU was not available to provide head orientation data, another source of heading information was necessary. Heading information was derived from the user's velocity, with an assumption being made that the user looks in the same direction that they are walking. For instance, if the user's velocity is 3 m/s north and 0 m/s east, the user's heading is calculated as due north (0 degrees). The spacing between boxes, and the size of the boxes, was also changed. Initial testing determined that for the velocities and distances involved in testing a person walking on the ground, 10 meter boxes placed every 100 meters did not provide sufficient information for effective navigation. The box sizes were changed to 5 meters per side, and placed every 20 meters. The software also omitted any path-recalculating algorithms, since a user on the ground can easily walk back to the path if they should happen to move away from it.

5.3. Path Configuration

The paths used for testing were created by marking locations with a GPS receiver. Each location was visited and marked after checking that the GPS had a good position solution. Each of the marked locations was designated as a waypoint; each path was made up of a number of waypoints that had to be visited sequentially. The layout of the paths can be seen in Figure 29.

Path 1 and Path 2 was located in a large open field, relatively free of obstacles. However, a few trees were present, obscuring the user's view of the path ahead. In addition, several small trenches crossed the field. Users had to remain aware of their surroundings to avoid the obstacles. Path 1 was the longest path in the test at 1207 meters, nearly twice as long as the next-longest path. Path 1 contained a tricky hairpin turn immediately after the beginning. This hairpin turn was designed to ensure that users could determine the path that they were on, and not head back to the starting waypoint. Path 2 was located in the same general area of Path 1. Path 2 was designed to be relatively straight and simple to navigate. It is slightly longer than Path 3, being 618 meters long. There was only one sharp turn in the path near the end.

Path 3 was located in a more crowded area, with buildings blocking many walking paths. The user had to navigate around buildings in order to arrive at each waypoint, since the straight-line path went through the buildings, which were designated as off limits. Path 3 was laid out as a loop, forcing the user to determine exactly where the path is in order not to repeat a segment that he had already navigated. This path is the shortest path presented to the user at only 534 meters of total length.



Path 1
Total Length: 1207 m

Path 2
Total Length: 618 m

Path 3
Total Length: 534 m

Figure 29: Path Layouts Used For Testing. Green Circles Indicate Start Points; Red Circles Indicate Stop Points.

5.4. Test Results

The primary performance metric was the time taken to walk each course with a given navigational reference. Time began when the user left the start waypoint and stopped when the user arrived at the last waypoint. The data gathered is listed in Table 1.

Table 1: Time Taken By Each User To Walk Each Path

User	Path 1 (GPS)	Path 2 (GPS)	Path 3 (GPS)	Path 1 (HUD)	Path 2 (HUD)	Path 3 (HUD)
1	13:37	6:57	N/A	14:08	7:15	N/A
2	15:20	7:40	N/A	15:07	8:16	N/A
3	14:23	7:21	N/A	15:41	7:24	N/A

Data was not collected for Path 3, either with the GPS or the HUD. When users attempted to navigate Path 3 with a HUD, it was observed that the proximity of the waypoints to each other, combined with the overlapping tunnels, resulted in a path that was very cluttered and hard to understand. None of the users tested were able to complete the course using the HUD due to the disorienting display. Therefore, the data for Path 3 was discarded.

Figure 30 is an overlay of the user trajectories for Path 1. The blue and gray dotted line represents the straight line path between waypoints. The red lines indicate the trajectories of users wearing the HUD; green lines represent the trajectories of users navigating with the GPS as a reference. The green and red dots show the beginning and end of the path, respectively. Only paths for two users are shown. The path of the third user was unable to be retrieved from the GPS receiver recording the testing. As can be seen, users with both systems were able to stay fairly close to the straight-line path. One user with the GPS did stray far from the straight-line path, but this is because the user chose to walk along a sidewalk rather than through the grass to the waypoint.

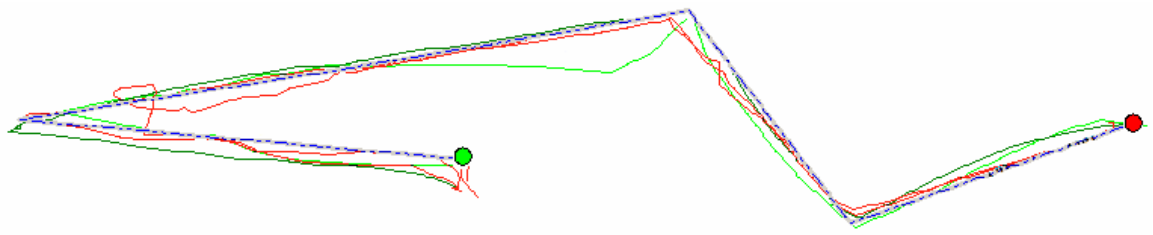


Figure 30: User Trajectories For Path 1. Green Lines Represent GPS-Based Paths And Red Lines Represent HUD-Based Paths. The Blue-And-Gray Dashed Line Indicates The Straight Line Path Between Waypoints. The Green Circle Represents The Starting Waypoint And The Red Circle Represents The Ending Waypoint.

Figure 31 shows an overlay of all user trajectories for Path 2. Blue lines indicate the trajectories for users with the HUD, and green lines indicate the trajectories of users with the GPS. Users were able to remain close to Path 2, no matter the guidance methodology used.

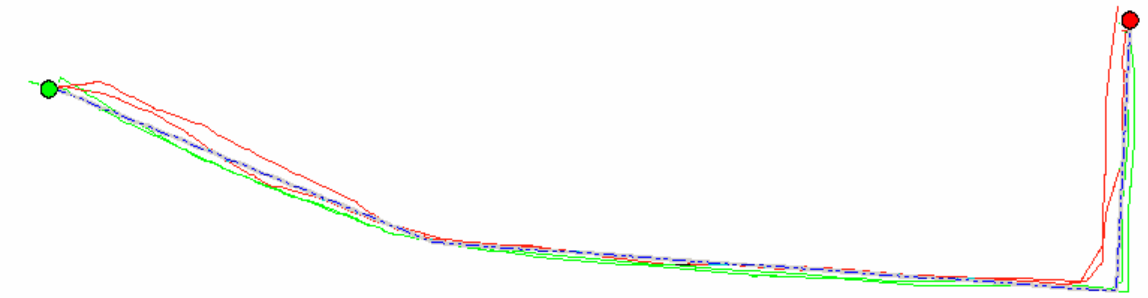


Figure 31: User trajectories for Path 2. Green Lines Represent GPS-Based Paths And Blue Lines Represent HUD-Based Paths. The Blue-And-Gray Dashed Line Indicates The Straight Line Path Between Waypoints. The Green Circle Represents The Starting Waypoint And The Red Circle Represents The Ending Waypoint

One interesting side effect of using the HUD for navigation is that users tend to cut corners in the path, as seen in Figure 32. Although the trajectories of the users brought them close to the waypoint, the users tended to ignore the waypoint and continue navigating, because they could see the path turning ahead of time.

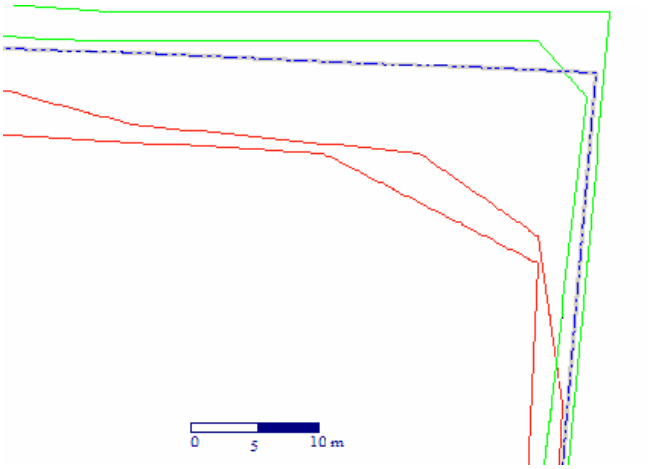


Figure 32: Users With The HUD (Red Lines) Tended To Cut Corners At Turns In The Path (Blue And Gray Line).

5.5. Analysis

Average times to navigate a course for each user are given in Table 2. Test subjects were able to complete a given path an average of 25 seconds quicker with the GPS navigational reference than the HUD navigational reference. This is a significant difference in course times.

Table 2: Average Times For A User To Navigate A Path.

User	Average Time (GPS)	Average Time (HUD)
1	10:17	10:41
2	11:30	11:41
3	10:52	11:32

This difference in efficiency is due to several possible reasons. The HUD in the test determined its field of view based on the user's velocity – whichever direction the user was walking in was assumed to be the direction the user was viewing. At corners in the path, the tunnel would move out of the user's field of view. This caused the user to become disoriented, and sometimes they had to spend time walking in circles in order to find the path again. The small field of view of the HUD exacerbated the problem. Integrating the HUD with a head tracker, so that the user could look around, would solve this problem.

Another cause of the HUD's problems stems from jumps in the GPS position (probably due to the gain or loss of reception to a GPS satellite). Several of the test subjects reported that their view of the virtual world 'jumped' around. One moment, they

were standing to the left of the path, and the next, they were standing to the right of the path. This made it difficult to stay on path and home in on the targets. If the jump was great enough, the test subjects had to search for the path.

The problem with using only GPS data for determining the user's heading is shown by the track in Figure 33. A zoomed in portion of Path 1's hairpin turn is shown, with black arrows indicating the direction of travel by the users. The green track is of a user equipped with the HUD accurately navigating to the waypoint. The red track shows another user with the HUD attempting to navigate to the waypoint. This user lost sight of the path, and began walking towards another segment of the path. When he passed it, he walked in a circle, searching for the path. Although the user eventually regained the correct path, he did spend extra time walking in the circle. If the user had been equipped with head tracking hardware, it is possible that he would have been able to see the waypoint without being forced to waste time walking in circles.

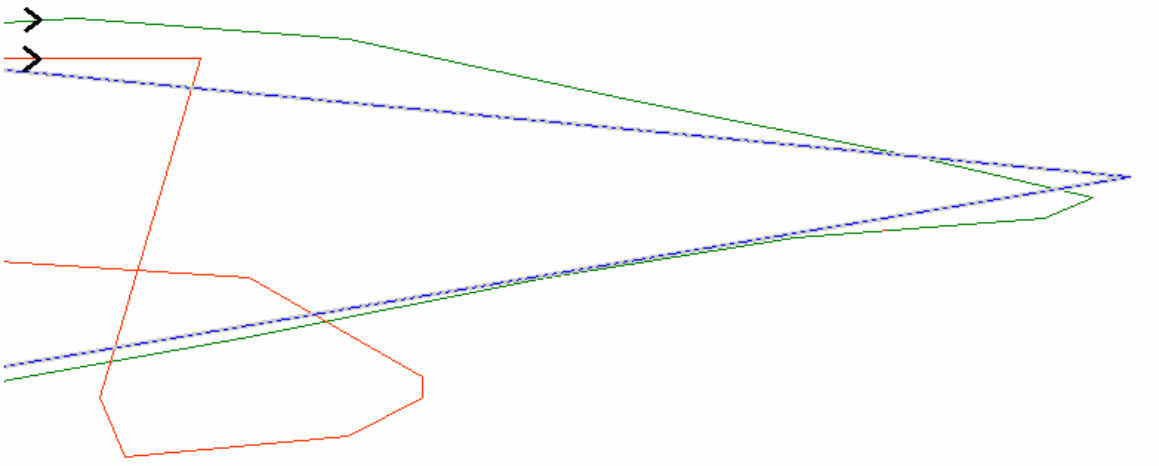


Figure 33: Track Of A User With HUD Losing Sight Of The Path. The Black Arrows Indicate The Direction Of Travel. The Green Line Represents The Trajectory Of A User With GPS. The Red Line Represents The Trajectory Of A User With The HUD. The Blue And Gray Line Represents The Path.

It was also noted that the repeatability of the GPS-reported altitude was poor. Initial surveying of the waypoints placed them at about 300 meters above sea level. During testing, the GPS often under- or over-estimated the user's altitude. Users complained that they were not level with the tunnel. When approaching waypoints, the sphere representing their goal would often disappear above or under their field of view. Both of the main causes of slow navigation were further compounded by the rate that the screen was updated. The GPS data came in at the rate of 1 Hz. When the user approached a waypoint or veered off course, they would begin turning in order to attempt to regain the course. However, a person can turn quite a bit in one second. Users would often overshoot their intended path, and waste a few seconds attempting to re-center the path or waypoint in their view.

All of these problems stem from the lack of a head tracker, combined with the narrow field of view available with the HUD. The head tracker would allow the user to constantly monitor their position relative to the path and the waypoints by looking at the part of the path that they need without having to turn their entire body and waiting for the GPS to update itself. The head tracker software could also implement a Kalman filter to smooth out the GPS's reported position, which would reduce the amount of jitter experienced by the user. Finally, the head tracker could reduce the latency in updating the screen which should improve user performance.

5.6. Conclusion

Testing of the system was conducted on the ground. Due to many hardware problems with the system under test originally specified, a different system, including

computer, GPS sensor, and software, was tested. Three users each walked three paths, and data was collected on their position and time to walk the course.

It was found that complex, short paths cause the HUD to display a cluttered, confusing display unsuitable for navigation. None of the test subjects were able to complete Path 3, and so the data for that path was discarded. For the paths that were completed, test subjects completed the course 25 seconds faster with the baseline navigational reference than with the graphical HUD. It is hypothesized that the lack of head-tracking hardware resulted in the poor performance of the HUD.

6. Conclusions and Recommendations

The system developed and discussed in this thesis represents a new navigational tool with the potential to greatly enhance the safety and capability of the USAF Pararescuemen and other paratroopers during HAHO jumps. Although a fully functional system, including a head tracker, could not be developed, a basic three-dimensional HUD was developed. This HUD used an algorithm to generate a path for a paratrooper given an LZ, wind conditions, and parachute performance characteristics. The output of this algorithm was in turn used by the Allegro graphics library to generate a three-dimensional HUD and displayed to the user.

Because of the lack of head-tracking hardware, a ground-based testing approach was taken to validate the concept of the HUD. It took users an average of 25 seconds longer to navigate the course using the HUD than compared to the baseline test of simply using a GPS unit displaying an arrow to the next waypoint. This difference in speed is explained by the lack of a head tracker integrated with the HUD, jitter in the user's reported position caused by GPS inaccuracies, and the delay in updating the screen caused by the 1Hz GPS updates.

6.1. Conclusions of Research

The research conducted achieved two objectives: designing and implementing a HUD suitable as a primary navigational reference for paratroopers conducting HAHO jumps, and testing the efficiency of that HUD. The first objective was quite successfully achieved. A PGA to generate a path was created and implemented. The HUD was

designed and implemented in Allegro. The HUD contains all elements necessary for a successful HAHO jump.

The second objective, testing the HUD, was not completely met. Difficulties in obtaining hardware prevented testing from occurring until late in the research. When the system was tested, a radically different version of the system than the one originally envisioned was used. The testing determined that the graphical HUD resulted in slower navigation times than the baseline navigational reference. This indicates that the graphical HUD is not as efficient as the baseline reference. However, it is felt that integrating the HUD with a head-tracker to allow more freedom in viewing the virtual world would greatly improve the system's performance.

6.2. Significance of Research

The research conducted for this thesis is significant in several ways, including the development of a HUD suitable for HAHO navigation, the creation of a path generation algorithm, and the testing of the system.

The development of the HUD is a novel approach to guiding a paratrooper. Existing paratrooper navigation aids do not aid the paratrooper in navigating under poor visibility conditions. By creating a HUD that shows the paratrooper where to go, even when he cannot see his destination, many more opportunities to complete missions are available. Cloud cover will no longer be an obstacle to paratroopers.

Crucial to the HUD is the PGA. The main advantage of a HAHO jump is the standoff range that it allows the paratroopers. By creating an algorithm to maximize the stand-off range, the key strength of a HAHO jump can be taken advantage of. The

current PGA uses a greedy algorithm to build a path from the ground up. Taking advantage of the mobility of the parachute, combined with the effect of the wind upon the parachute, makes the PGA more realistic. Unfortunately, testing of the PGA's abilities could not be conducted before the completion of this research.

Finally, the HUD was tested on the ground. Although the testing showed that users were able to navigate a path quicker with a baseline GPS reference, the HUD showed promise. The testing revealed that integrating a head tracker with the system is a key step to complete before a final version can be fielded, as the current method of determining heading is unsatisfactory for the user. Once a viable head tracker is created, the HUD will be much more usable and easier to understand.

6.3. Recommendations for Future Research

There are several avenues of this project worth further investigation: improving the path predicting algorithm (as described in Section 3.5.3), integrating the HUD system with an IMU for head tracking, and improving the acceptability of the hardware for field use. There are also additions that could be made to the system that could expand the capabilities of paratroopers.

Due to hardware unavailability, integrating the HUD with an IMU for head tracking proved to be impossible during the execution of this thesis. Head tracking would make this project much more useful in the real world. Integrating an effective head tracking component to this system should increase the rate at which a user could navigate a course, and make the system more acceptable to the user. Also, head tracking

would enable researchers to conduct more realistic testing of the system, such as conducting actual parachute jumps at different altitudes.

Research must also be conducted into the hardware that the HUD software runs on. Currently, the system weighs a few pounds, and is somewhat bulky. Battery life is somewhat limited, and the PC-104 can become extremely hot during prolonged use. The system is also not hardened against inclement weather, which is where the HUD system would be deployed. Minimizing the power usage, size, and heat output of the system while maximizing ruggedness and user friendliness will be a priority before this system can be considered operational. Making the PC-104 small and power efficient enough to be acceptable in field conditions may prove to be impossible; alternative methods of implementing the hardware (such as an ARM processor) should be investigated.

The system currently takes input such as the wind profile and parachute data from files manually entered into the system. An ideal system would interface with the aircraft carrying the paratroopers and be able to automatically download wind data, landing zone information, and other important data. Research should be conducted into methods that could allow this to happen.

The system's performance may be further improved by measuring the time that the system takes to generate a single graphical frame. This time consists of the time taken to read in the user's position, determine the orientation of the user's head, determine if the user is 'far away,' calculate the graphical image to be displayed, and display the final image. If the system is spending significant amounts of time waiting for more input from the ISS, the PGA or the 'far away' algorithm's fidelity might be able to

be improved. For instance, instead of making a simple distance calculation during the 'far away' check, the system could simulate the rest of the paratrooper's jump from his current position, using the collected wind data. If the simulated jump indicates that the paratrooper cannot make it to the LZ, then he is 'far away.'

Another avenue for further research would be the applicability of this system to other types of parachute jumps, such as High Altitude, Low Opening (HALO) jumps, or regular static-line jumps.

Regardless of the format of the jump, paratroopers must land with their teammates. If a communications system could be integrated with the HUD, positions of other paratroopers involved in the jump could be displayed. This would enable teams to remain together, even if visibility conditions make visual contact impossible.

Appendix A – HUD Source Code Used for Testing

```
/*-------------------------------------------------------------------------------
 * HUD Software
 * 2Lt. Jason Thompson
 * AFIT/ENG/GCS/05M
 * This software is intended to read in some external positioning information
 * and display a heads up display.  This code should not be used for any
 * operational mission.  This code is intended for research use only.  This code
 * may be copied and modified freely.
 *-------------------------------------------------------------------------------
 */

/*-------------------------------------------------------------------------------
 * All #includes go here.
 *-------------------------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <allegro.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>

/*-------------------------------------------------------------------------------
 * All #defines go here.
 *-------------------------------------------------------------------------------
 */

#define X 0
#define Y 1
#define Z 2
#define LAT 0
#define ALT 1
#define LON 2
#define YAW 0
#define PITCH 1
#define ROLL 2

#define YES 1
#define NO 0
#define OCCUPIED 1
#define EMPTY 0
#define READ 1
#define WRITE 0

#define PI 3.14159265                   //The value of PI
#define RADIUS 6378137                  //The radius of the earth in meters

#define BOXSIZE 5                       //The size of the sides of a box in meters
#define BOXDISTANCE 25                  //Distance between boxes in meters
#define NUMWP 30                        //The number of waypoints that are allowed
#define BUFSIZE 20                      //Data buffer size
#define LANDINGHEIGHT 100               //The altitude (in meters) of when we turn
                                            //to land
#define BPS 60                          //How many 'beats per second' this runs at
#define HEADINGWIDTH 80                 //How many degrees are shown in our
                                            //heading indicator
#define VISITDISTANCE 5              //How close we have to get to a waypoint in
                                     //order to mark it as visited
```

```
#define BLACK 0,0,0                     //Allegro code for black
#define WHITE 255,255,255               //Allegro code for white
#define RED 255,0,0                     //Allegro code for red
#define GREEN 0,255,0                   //Allegro code for green

#define WIDERES 640                     //Resolution of our screen
#define HIGHRES 480                     //Resolution of our screen

#define DEG(n) ((n) * 128.0 / PI)       //Macro to convert from radians to 256 deg
                                        //Allegro uses 256 degrees in a circle


/*--------------------------------------------------------------------------------
 * Structures
 *--------------------------------------------------------------------------------
 */
struct navDataStruct {
    double datetime;                        //The date and time - yymmddhhmmss

    double currentLatitude;         //Should be in format dd.xxxxxx
                                        //North is positive

    double currentLongitude;        //Should be formatted as dd.xxxxxx
                                        //East is positive

    double currentAlt;               //Altitude in meters

    double velocity[3];              //Velocity in each dimension
                                    //[0] = east/west velocity (east is +)
                                    //[1] = vertical velocity (up is +)
                                    //[2] = north/south velocity (north is +)

    float trackangle;               //The track angle - from 0-359

    short HDOP;                     //The horizontal DOP
    short VDOP;                     //The vertical DOP

    bool last;                      //Sort of a checksum - set to TRUE if the data
                                        //is finished sending from the GPS

};


struct wpStruct {

    int idNum[NUMWP];               //The waypoint's ID number
    double LLA[NUMWP][3];           //The waypoint's latitude, longitude, and alt
    V3D_f XYZ[NUMWP];               //The waypoint's XYZ coordinates
    int occupied[NUMWP];            //Is this waypoints occupied or not?
    int visited[NUMWP];             //Has this waypoint been visited?
    double bearing[NUMWP];          //The bearing to this waypoint

};

struct windDataStruct{

    int lowAlt[NUMWP-2];            //The starting alt for which this data is valid
    int highAlt[NUMWP-2];           //The ending alt for which this data is valid
    int direction[NUMWP-2];         //The direction the wind is blowing FROM
                                        //(360 degrees)
    float speed[NUMWP-2];           //The speed that the wind is blowing at
                                        //(meters per second)
    int occupied[NUMWP-2];          //Is this wind data segment occupied?
};

struct parachuteDataStruct{
    float forwardSpeed;             //This is the parachute's forward velocity
```

```
                                        //in meters per second
    float rateOfDescent;            //This is the parachute's rate of descent
                                        //in meters per second
};

/*-------------------------------------------------------------------------------
 * Function Prototypes
 *-------------------------------------------------------------------------------
 */

int infoMain(void);                         //The main info gathering function
int infoGetBearingNextWaypoint(void);       //Gets the bearing of the next waypoint
int infoGetCommFilePipe(void);              //Gets a file pipe for the comm port
int infoGetLZInfo(void);                    //Gets the LZ information
int infoGetParachuteCharacteristics(void);  //Gets the parachute information
int infoGetWindInfo(void);                  //Gets the wind information
void infoReadPositionData(int);             //Reads data from the GPS/IMU
void infoUpdatePosition(int);               //Updates the user's position and
                                                //head orientation (viewing angle)
void infoWindInfoCount(void);               //Counts how many data points we have
                                                //for the wind info
void infoWayPointCount(void);               //Counts how many waypoints are present
void infoWayPointInit(void);                //Init the waypoint info
int infoWPFileRead(void);
void infoVisitedWaypoints(void);            //Updates whether or not we have
                                                //visited the waypoints.

void gfxDrawAltitude(BITMAP *);             //Draws the altitude scale on right
void gfxDrawArrows(BITMAP *, MATRIX_f);     //Draws guidance arrows if WP is
                                                //not visible
void gfxDrawBoxes(BITMAP *, MATRIX_f, MATRIX_f);//Draws the boxes that make up
                                                //the path
int gfxDrawCircles(BITMAP *, MATRIX_f, int);//Draws the waypoint circles
void gfxDrawHeadingBody(BITMAP *);          //Draws the heading scale at top
int gfxVerify(BITMAP *);                    //Draws the route overview (not used)
void gfxRender(BITMAP *);                   //Main drawing loop - called to draw
                                                //a new frame


void distCalcDistance(void);                //Calculates the WPs' distance from LZ
double distXDistance(double, double, double);//Calcs the east/west distance from LZ
double distYDistance(double, double);       //Calcs the altitude difference from LZ
double distZDistance(double, double);       //Calcs the north/south distance fromLZ
double calcDistBetweenPoints(double, double, double);
double calcHeading(void);                   //Calcs the heading

void timerLogicCounter(void);       //Timer function that ensures the program runs
                                        //at the right speed - change #define BPS
                                        //to control how many frames per second
                                        //at most that the program runs at

/*-------------------------------------------------------------------------------
 * Global Variables
 *-------------------------------------------------------------------------------
 */

windDataStruct WindInfo;                //Data structure to hold all wind info
wpStruct wayPoints;                     //Data structure to hold all waypoints
parachuteDataStruct parachuteData;      //Data structure to hold parachute info
navDataStruct currentData;              //Data to hold our current position and
                                            //head orientation
navDataStruct readData;                 //Data that is read in from the GPS/IMU
navDataStruct previousData;             //Data that holds positioninfo after we
                                            //finish using it

int windInfoCount;                      //How many wind levels we have
int wayPointCount;                      //How many waypoints are present
```

```
volatile int logicCounter = 0;          //This is used for the timer

int activewaypoint = 1;                 //Which waypoint we are headed for now
double distNextWP;                                      //The distance to the next wp

int viewport_w = WIDERES;                   //The width of our viewport
int viewport_h = HIGHRES;                   //The height of the viewport
                                            //Both viewports should be set to
                                            //the corresponding screen resolution
int fov = 48;                           //Our field of fiew (degrees)
float aspect = 1;                       //The aspect ratio of our view
double xpos = 0;                         //The user's east/west position
double ypos = 0;                         //The user's altitude
double zpos = 0;                         //the user's north/south position
double headingView = 0;                  //Which way we are looking
double headingTemp = 0;                  //Just a temp variable - may not be used
double headingBody = 0;                  //Which way we are heading
double pitch = 0;                        //The pitch of our view (looking up or down)
double roll = 0;                         //How our head is tilted
double xfront = 0, yfront = 0, zfront = 0;//In-front vector (which way camera is
                                            //looking)

/*---------------------------------------------------------------------------
 * Main function ties everything together
 *---------------------------------------------------------------------------
 */
void main (void)
{
    /* Declare variables */
    int success = FALSE, i = 0, commFD;
    BITMAP *buffer;

    /* Initialize things */
    allegro_init();             //Initialize the Allegro library
    install_keyboard();         //Initialize the Allegro keyboard handler
    install_timer();            //Initialize the Allegro timer routines
    infoWayPointInit();         //Initialize the Waypoint information

    srand(time(NULL));

    /* Set up the Allegro Timer to beat 60 times a second */
//  install_int_ex(timerLogicCounter, BPS_TO_TIMER(BPS));
//  LOCK_VARIABLE(logicCounter);
//  LOCK_FUNCTION(timerLogicCounter);


    /* Gather all Information */
    success = infoMain();
    commFD = infoGetCommFilePipe();
    if(commFD == -1)
        success = FALSE;

    /* Print out a list of our waypoints coordinates */
    for(i = 0; i < wayPointCount; i++)
    {
        printf("Waypoint %d: %f, %f, %f\n", i,
            wayPoints.XYZ[i].x, wayPoints.XYZ[i].y, wayPoints.XYZ[i].z);
    }

    /* Initialize the graphics */
    if(success == TRUE)
    {
        /* Sets up the allegro safe graphics mode */
        if(set_gfx_mode(GFX_SAFE, WIDERES, HIGHRES, 0, 0) != 0)
        {
            set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
```

```
            allegro_message("Unable to set a graphics mode!\n%s\n", allegro_error);
            success = FALSE;
        }
    }


    if(success == TRUE)
    {
        /* Create a bitmap to draw everything to, and clear it to black */
        set_palette(desktop_palette);

        buffer = create_bitmap(SCREEN_W, SCREEN_H);

        clear_to_color(screen, makecol(BLACK));
        clear_to_color(buffer, makecol(BLACK));
    }


    /* Initialize the user's position to the last waypoint */
    xpos = wayPoints.XYZ[wayPointCount-1].x;
    ypos = wayPoints.XYZ[wayPointCount-1].y;
    zpos = wayPoints.XYZ[wayPointCount-1].z;

    /* Enter the main rendering loop */
    while(!key[KEY_ESC])
    {
    //    while(logicCounter > 0)
    //    {
            /* Render the scene and get user's position*/
            gfxRender(buffer);
            infoUpdatePosition(commFD);


            /* Wait until we can draw to the screen again    */
            /* This is how we implement the beats per second */
            /* logic from allegro.  Limits framerate on fast */
            /* computers                                     */
    //        logicCounter--;
    //    }

        /* Finally, draw the bitmap to the video memory */
        blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);
    }


    /* Clean up everything and quit */
    allegro_exit();


    return;
}
END_OF_MAIN()

/*--------------------------------------------------------------------------------
 * Information Gathering Section
 * The functions in this section of code are responsible for gathering information
 * from the user.
 * The three main pieces of information are:
 * -Parachute characteristics
 * -Landing zone information
 * -Wind information
 *--------------------------------------------------------------------------------
 */

int infoMain(void)
{
    int success = FALSE, i = 0;
```

```
    /* Get the waypoint information */
    success = infoWPFileRead();

    /* Find out how many waypoints have been read in */
    for(i = 0; i < NUMWP; i++)
    {
        if(wayPoints.occupied[i] == OCCUPIED)
            wayPointCount++;
    }

    /* Convert lat/lon into XYZ coordinates */
    for(i = 0; i < wayPointCount; i++)
    {
        wayPoints.XYZ[i].x = distXDistance(wayPoints.LLA[i][LAT],
            wayPoints.LLA[i][LON], wayPoints.LLA[wayPointCount-1][LON]);

        wayPoints.XYZ[i].y = -distYDistance(wayPoints.LLA[i][ALT], 0);

        wayPoints.XYZ[i].z = distZDistance(wayPoints.LLA[i][LAT],
            wayPoints.LLA[wayPointCount-1][LAT]);

    }

    return success;
}


/*-----------------------------------------------------------------------------
 * infoGetBearingNextWaypoint
 * This calculates the bearing of the next waypoint.
 *-----------------------------------------------------------------------------
 */
int infoGetBearingNextWaypoint(void)
{
    int bearingNextWaypoint;

    double lat2, lat1, lon2, lon1;
    double x, y;

    /* Find out what is the next waypoint */

    /* Set up our current position and the location of the place we want to get */
    /* the bearing to                                                           */
    lat2 = wayPoints.LLA[activewaypoint][LAT];
    lon2 = wayPoints.LLA[activewaypoint][LON];

    lat1 = currentData.currentLatitude;
    lon1 = currentData.currentLongitude;

    /* If we are not at our next waypoint, calculate the bearing    */
    /* I used a formula I found on the web                          */

    if(lat1 != lat2 && lon1 != lon2)
    {
        y = sin(lon2 - lon1) * cos(lat2);
        x = cos(lat1) * sin(lat2) - sin(lat1) * cos(lat2) * cos(lon2-lon1);

        if(y > 0)
        {
            if(x > 0)
                bearingNextWaypoint = (int)DEG(atan(y/x));
            if(x < 0)
                bearingNextWaypoint = 180 - (int)DEG(atan(-y/x));
            if(x == 0)
                bearingNextWaypoint = 90;
        }
```

```
        if(y < 0)
        {
            if(x > 0)
                bearingNextWaypoint = (int)DEG(-atan(-y/x));
            if(x < 0)
                bearingNextWaypoint = (int)DEG(atan(y/x)) - 180;
            if(x == 0)
                bearingNextWaypoint = 270;
        }

        if(y == 0)
        {
            if(x > 0)
                bearingNextWaypoint = 0;
            if(x < 0)
                bearingNextWaypoint = 180;
            if(x == 0)
                bearingNextWaypoint = 0;
        }
    }
    else
        bearingNextWaypoint = 0;

    return bearingNextWaypoint;

}

/*-------------------------------------------------------------------------------------
 * infoGetCommFilePipe
 *-------------------------------------------------------------------------------------
 */

int infoGetCommFilePipe (void)
{
    int fd; /* file descriptor for the serial port */

    /* Open the file descriptor */
    fd = open("/dev/ttyS0", O_RDONLY | O_NOCTTY);
    if(fd == -1)
    {
        perror("open_port: Unable to open /dev/ttyS0");
    }

    fcntl(fd, F_SETFL, 0);

    return fd;
}

/*-------------------------------------------------------------------------------------
 * infoWPFileRead
 * Reads in the list of waypoints from a file
 *-------------------------------------------------------------------------------------
 */
int infoWPFileRead(void)
{

printf("entering infowpfileread\n");
    int success = TRUE, i = 0, j = 0;                   //These are counters;
    char c;
    char filedata[BUFSIZE];

    FILE *wpFilePtr;

    /* Attempt to open the file; if it is not found, set success to FALSE */
    if((wpFilePtr = fopen("wp.dat", "r")) == NULL)
    {
```

```
    printf("Could not open wp.dat!  Please make sure that the file exists.\n");
    success = FALSE;
}

/* If the file is opened successfully, read in data*/
else
{
    while (i < NUMWP && !feof(wpFilePtr))
    {
        c = fgetc(wpFilePtr);
        switch(c)
        {
        /* If the line starts with #, ignore it, as it is a comment */
        case '#':
            while(c != '\n')
                c = fgetc(wpFilePtr);
            break;

        /* Ignore newlines */
        case '\n': case '\r':
            break;

        /* If we get a number.. do something */
        case '0':case '1':case '2':case '3':case '4':case '5':case '6':case '7'
            :case '8':case '9': case '-':
            /* Get the Latitude */
            while(c != ' ' && j < BUFSIZE)
            {
                filedata[j] = c;
                c = fgetc(wpFilePtr);
                //printf("%c", c);
                j++;
            }

            wayPoints.LLA[i][LAT] = atof(filedata);
            for(j = 0; j < BUFSIZE; j++)
            {
                filedata[j] = ' ';
            }
            j = 0;

            c = fgetc(wpFilePtr);

            /* Get the Longitude */
            while(c != ' ' && j < BUFSIZE)
            {
                filedata[j] = c;
                c = fgetc(wpFilePtr);
                //printf("%c", c);
                j++;
            }

            wayPoints.LLA[i][LON] = atof(filedata);
            for(j = 0; j < BUFSIZE; j++)
            {
                filedata[j] = ' ';
            }
            j = 0;

            c = fgetc(wpFilePtr);

            /* Get the Altitude */
            while(c != '\n' && j < BUFSIZE)
            {
                filedata[j] = c;
                c = fgetc(wpFilePtr);
                ////printf("%c", c);
```

```
                    j++;
                }

                wayPoints.LLA[i][ALT] = atof(filedata);
                for(j = 0; j < BUFSIZE; j++)
                {
                    filedata[j] = ' ';
                }
                j = 0;

                /* Print a confirmation that we got the waypoint correctly and */
                /* Then set the waypoint.occupied flag, so we know we have a wp*/
                printf("%f, %f, %f\n", wayPoints.LLA[i][LAT], wayPoints.LLA[i][LON],
                    wayPoints.LLA[i][ALT]);
                wayPoints.occupied[i] = OCCUPIED;
                i++;
                break;

            default:
                /* If we reached the end of the file, we are good. */
                if(feof(wpFilePtr))
                    printf("\nEnd of file reached!  %d waypoints read.\n", i);
                else
                {
                    printf("Something went wrong.\n");
                    printf("%c\n", c);
                    success = FALSE;
                }
                break;
            }

        }
    }

    fclose(wpFilePtr);


    return success;
}



/*-----------------------------------------------------------------------------
 * infoGetLZInfo
 * This function gathers the LZ's location from the user.  At the moment, it's
 * hardcoded to AFIT's location.  The LZ is unnecessary for the testing, but I left
 * it in here for when we can do parachute testing.
 * We should probably find a more elegant way to program the LZ's location
 * Perhaps read in a waypoint from the GPS, if it offers that functionality.
 *-----------------------------------------------------------------------------
 */
int infoGetLZInfo(void)
{
    int success = FALSE;

    wayPoints.LLA[0][LAT] = 39.786;
    wayPoints.LLA[0][LON] = -84.0915;
    wayPoints.LLA[0][ALT] = 256.36;
    wayPoints.occupied[0] = YES;

    /* Convert the Lat/lon/alt to XYZ coordinates */
    distCalcDistance();

    success = TRUE;

    return success;
}
```

```
/*-------------------------------------------------------------------------------
 * infoReadPositionData
 * This function reads data from the GPS/IMU.  Currently set up for text mode
 * transfers from a Garmin RINO 120.
 *-------------------------------------------------------------------------------
 */
void infoReadPositionData(int commFD)
{
    char buf[58], temp[15];
    char at[2];
    char datetime[13];
    char latitude[9];
    char longitude[10];
    char altitude[7];
    char eastvel[6];
    char northvel[6];
    char upvel[6];

    double ddtemp, mmtemp;

    buf[57] = '\0';
    at[1] = '\0';
    datetime[12] = '\0';
    latitude[8] = '\0';
    longitude[9] = '\0';
    altitude[6] = '\0';
    eastvel[5] = '\0';
    northvel[5] = '\0';
    upvel[5] = '\0';

    memcpy(&previousData, &currentData, sizeof(navDataStruct));

    /* Read data from the comm port - blocks until we receive data */
    while(buf[0] != '@')
    {
        read(commFD, buf, sizeof(buf)-1);
    }

    /* Parse out the relevant data strings */
    strncpy(at, &buf[0],sizeof(at)-1);
    strncpy(datetime, &buf[1], sizeof(datetime)-1);
    strncpy(latitude, &buf[13], sizeof(latitude)-1);
    strncpy(longitude, &buf[21], sizeof(longitude)-1);
    strncpy(altitude, &buf[34], sizeof(altitude)-1);
    strncpy(eastvel, &buf[40], sizeof(eastvel)-1);
    strncpy(northvel, &buf[45], sizeof(northvel)-1);
    strncpy(upvel, &buf[50], sizeof(upvel)-1);

        /* Copy over the time */
        currentData.datetime = atof(datetime);


        /* The lat/lon data is in dd.mm.mmmm format right now */
        /* We need to convert it to dd.dddddd format.                 */
        strncpy(temp, &latitude[1], 2);
        temp[2] = '\0';
        ddtemp = atof(temp);
        mmtemp = atof(&latitude[3]);
        mmtemp = mmtemp / 60000;
        ddtemp = ddtemp + mmtemp;

    if(latitude[0] == 'N')
        currentData.currentLatitude = ddtemp;
    else
        currentData.currentLatitude = 0 - ddtemp;
```

```
        /* Now convert the longitude */
    strncpy(temp, &longitude[1], 3);
    temp[3] = '\0';
    ddtemp = atof(temp);
        mmtemp = atof(&longitude[4]);
        mmtemp = mmtemp / 60000;
        ddtemp = ddtemp + mmtemp;

    if(longitude[0] == 'E')
        currentData.currentLongitude = ddtemp;
    else
        currentData.currentLongitude = 0 - ddtemp;

    /* Convert the altitude */
    if(altitude[0] == '+')
        currentData.currentAlt = atof(&altitude[1]);
    else
        currentData.currentAlt = atof(&altitude[1]);

    /* Convert east/west velocity */
    if(eastvel[0] == 'E')
        currentData.velocity[0] = atof(&eastvel[1]) / 10;
    else
        currentData.velocity[0] = 0 - atof(&eastvel[1]) / 10;

    /* Convert north/south velocity */
    if(northvel[0] == 'N')
        currentData.velocity[2] = atof(&northvel[1]) / 10;
    else
        currentData.velocity[2] = 0 - atof(&northvel[1]) / 10;

    /* Convert up down velocity */
    if(upvel[0] == 'U')
        currentData.velocity[1] = atof(&upvel[1]) / 10;
    else
        currentData.velocity[1] = 0 - atof(&upvel[1]) / 10;

    /* need to calculate our heading from our previous location and our new location*/
    //headingBody = calcHeading();
    headingBody += 0.05;
    if(headingBody > 2*PI)
       headingBody = 0.0;
    return;
}

/*-------------------------------------------------------------------------------
 * infoUpdatePosition
 * This function updates the user's positions and viewing angle.
 *-------------------------------------------------------------------------------
 */
void infoUpdatePosition(int fdPipe)
{
    FILE *trackFilePtr;

    infoReadPositionData(fdPipe);

    /* Convert from Lat/lon/alt coords to XYZ coords */
    xpos = distXDistance(currentData.currentLatitude, currentData.currentLongitude,
                wayPoints.LLA[wayPointCount-1][LON]);
    ypos = -distYDistance(currentData.currentAlt, 0);
    zpos = distZDistance(currentData.currentLatitude,
        wayPoints.LLA[wayPointCount-1][LAT]);

    /* This next part could be uncommented to record where we've been for future */
    /* analysis.  Just writes to a file our new position                          */
    if((trackFilePtr = fopen("history.txt", "a")) == NULL)
```

```
    {
        printf("Could not open history!  Please make sure that the file exists.\n");
    }

       fprintf(trackFilePtr, "%f\n", currentData.datetime);
       fprintf(trackFilePtr, "Lat, lon, alt:\n");
    fprintf(trackFilePtr, "%f %f %f\n", currentData.currentLatitude,
        currentData.currentLongitude, currentData.currentAlt);

    fprintf(trackFilePtr, "East/Up/North velocity, heading:\n");
    fprintf(trackFilePtr, "%f %f %f %f\n\n", currentData.velocity[0],
       currentData.velocity[1], currentData.velocity[2], headingBody);

    fclose(trackFilePtr);

    /* Update which waypoints we have visited */
    infoVisitedWaypoints();

    return;
}


/*-------------------------------------------------------------------------------
 * infoWayPointCount
 * This function counts how many waypoints are present
 *-------------------------------------------------------------------------------
 */
void infoWayPointCount(void)
{
    int i;

    wayPointCount = 0;

    /* Very simple - just iterate through the waypoint structure and count how  */
    /* many OCCUPIED flags we have.                                             */
    for (i = 0; i < NUMWP; i++)
    {
        if(wayPoints.occupied[i] == YES)
            wayPointCount++;
    }

    return;
}


/*-------------------------------------------------------------------------------
 * infoWayPointInit
 * This function initializes the waypoint structure to a clean state
 *-------------------------------------------------------------------------------
 */
void infoWayPointInit(void)
{
    int i, j;

    for (i = 0; i < NUMWP; i++)
    {
        wayPoints.idNum[i] = i;
        for(j = 0; j < 2; j++)
        {
            wayPoints.LLA[i][j] = 0.0;
        }
        wayPoints.XYZ[i].x = 0.0;
        wayPoints.XYZ[i].y = 0.0;
        wayPoints.XYZ[i].z = 0.0;
        wayPoints.occupied[i] = NO;
        wayPoints.visited[i] = NO;
        wayPoints.bearing[i] = 0.0;
```

```
    }

    return;
}


/*-------------------------------------------------------------------------------
 * infoVisitedWaypoints
 * This function calculates which waypoints have already been visited.
 * If we are lower than the waypoint, we have visited it.  Once it's been visited,
 * you can't 'unvisit' it.  Obviously, for waypoints on the ground, this algorithm
 * does not work.  For a parachutist, it does work.
 *-------------------------------------------------------------------------------
 */
void infoVisitedWaypoints(void)
{
    double dX, dZ;

        /* Calculate the distance */

        dX = distXDistance(currentData.currentLatitude, currentData.currentLongitude,
                wayPoints.LLA[activewaypoint][LON]);
        dZ = distZDistance(currentData.currentLatitude,
wayPoints.LLA[activewaypoint][LAT]);

        distNextWP = sqrt((dX*dX) + (dZ*dZ));


        if(distNextWP < VISITDISTANCE)
        {
                wayPoints.visited[activewaypoint] = TRUE;
                activewaypoint++;
        }


    return;
}


/*-------------------------------------------------------------------------------
 * Graphics Section
 * The functions in this section of code are responsible for generating all of the
 * graphics that will be displayed to the user.
 *-------------------------------------------------------------------------------
 */
/*-------------------------------------------------------------------------------
 * gfxDrawAltitude
 *
 * Function Description:
 * This function draws an altitude scale on the right hand side of the bitmap,
 * providing a graphical way to show the parachutist his altitude.
 * The arrow points to his current location.
 * The top bar is equal to the height of the highest waypoint
 * The bottom bar is equal to sea level (0 meters)
 * The small middle bar is the altitude of the final waypoint.
 *
 * There may be a cleaner, more noticable way to represent altitude to the PJ,
 * like a thermometer scale.  Need to do real life testing to see how visible
 * this scale is, and how functional it is.
 *
 * Function Inputs:
 * *bmp - pointer to a bitmap to draw all of this stuff to.
 *
 * Function Return:
 * None
 *
```

```
 *---------------------------------------------------------------------------
 */
void gfxDrawAltitude(BITMAP *bmp)
{

    char buffer[6];
    int topLine, bottomLine, altLine, LZLine;
    int i;

    double maxAltitude = 0.0;

    /* Find out which waypoint has the highest altitude */
    for(i = 0; i < wayPointCount; i++)
    {
        if(-wayPoints.XYZ[i].y > maxAltitude)
            maxAltitude = -wayPoints.XYZ[i].y;
    }

    /* This is our current altitude scaled to the max-altitude */
    //altLine = (int)(100 * -ypos) / (int)maxAltitude + 1;

    /* This is the altitude of the LZ */
    //LZLine = (int)(100 * wayPoints.LLA[0][ALT]) / ((int)maxAltitude + 1);


    /* The top of the line is 50 pixels above the middle of the screen  */
    /* The bottom line is 100 pixels below the top line                 */
    /* The altitude line is relative to the bottom line.  Basically, we */
    /* find a proportion of how high we are, relative to the max alt and*/
    /* LZ, and use that.                                                */

    topLine = SCREEN_H / 2 - 50;
    bottomLine = topLine + 100;
    altLine = bottomLine - altLine;
    LZLine = bottomLine - LZLine;

    /* Draw the framework of the altimeter      */
    /* Outline these things in black, as well   */
    line(bmp, SCREEN_W-1, topLine, SCREEN_W-1, bottomLine, makecol(WHITE));
    line(bmp, SCREEN_W-2, topLine, SCREEN_W-2, bottomLine, makecol(BLACK));

    line(bmp, SCREEN_W-1, topLine, SCREEN_W-6, topLine, makecol(WHITE));
    line(bmp, SCREEN_W-1, topLine-1, SCREEN_W-7, topLine-1, makecol(BLACK));
    putpixel(bmp, SCREEN_W-7, topLine, makecol(BLACK));
    line(bmp, SCREEN_W-2, topLine+1, SCREEN_W-7, topLine+1, makecol(BLACK));

    line(bmp, SCREEN_W-1, bottomLine, SCREEN_W-6, bottomLine, makecol(WHITE));
    line(bmp, SCREEN_W-1, bottomLine+1, SCREEN_W-7, bottomLine+1, makecol(BLACK));
    putpixel(bmp, SCREEN_W-7, bottomLine, makecol(BLACK));
    line(bmp, SCREEN_W-2, bottomLine-1, SCREEN_W-7, bottomLine-1, makecol(BLACK));

    /* Draw the line that indicates where the LZ is     */
    /* And outline in black                             */
    line(bmp, SCREEN_W-1, LZLine, SCREEN_W-5, LZLine, makecol(WHITE));
    line(bmp, SCREEN_W-2, LZLine+1, SCREEN_W-6, LZLine+1, makecol(BLACK));
    line(bmp, SCREEN_W-2, LZLine-1, SCREEN_W-6, LZLine-1, makecol(BLACK));
    putpixel(bmp, SCREEN_W-6, LZLine, makecol(BLACK));

    /* Draw the current altitude arrow  */
    /* And outline in black             */
    line(bmp, SCREEN_W-1, altLine, SCREEN_W-6, altLine - 5, makecol(WHITE));
    line(bmp, SCREEN_W-2, altLine-2, SCREEN_W-6, altLine-6, makecol(BLACK));
    line(bmp, SCREEN_W-7, altLine-6, SCREEN_W-7, altLine-4, makecol(BLACK));
    line(bmp, SCREEN_W-2, altLine, SCREEN_W-6, altLine-4, makecol(BLACK));

    line(bmp, SCREEN_W-1, altLine, SCREEN_W-6, altLine + 5, makecol(WHITE));
    line(bmp, SCREEN_W-2, altLine+2, SCREEN_W-6, altLine+6, makecol(BLACK));
```

```
    line(bmp, SCREEN_W-7, altLine+6, SCREEN_W-7, altLine+4, makecol(BLACK));
    line(bmp, SCREEN_W-3, altLine+1, SCREEN_W-6, altLine+4, makecol(BLACK));

    /* Draw a line indicating where the LZ is */
    line(bmp, SCREEN_W-1, LZLine, SCREEN_W-5, LZLine, makecol(WHITE));

    /* Print the user's altitude and stick it on the bitmap */
    sprintf(buffer, "%d", (int)-ypos);
    textout_right(bmp, font, buffer, SCREEN_W - 1, topLine - 10, makecol(WHITE));


    return;
}
/*-------------------------------------------------------------------------------
 * gfxDrawArrows
 *
 * Function Description:
 * This function will draw an arrow pointing to the next active waypoint, if the
 * waypoint is not visible.
 *
 * Function Inputs:
 * *bmp - pointer to a bitmap to draw all of this stuff to.
 * *camera - a transformation matrix that contains information about the camera's
 * viewing angle
 *
 * Function Return:
 * None
 *
 *-------------------------------------------------------------------------------
 */
void gfxDrawArrows(BITMAP *bmp, MATRIX_f *camera)
{


    return;
}
/*-------------------------------------------------------------------------------
 * gfxDrawBoxes
 *
 * Function Description:
 * This function draws the boxes between waypoints; these make up the path.
 *
 * Function Inputs:
 * *bmp - pointer to a bitmap to draw all of this stuff to.
 * *camera - a transformation matrix that contains info about the camera's
 * viewing angle
 * *rotation - a transformation matrix that specifies how to rotate the box so
 * it looks right
 *
 * Function Return:
 *
 *-------------------------------------------------------------------------------
 */
void gfxDrawBoxes (BITMAP *bmp, MATRIX_f *camera, MATRIX_f *rotation)
{
    /* Arrays to hold our vertexes */
    V3D_f _v[4], _vout[8], _vtmp[8];
    V3D_f *v[4], *vout[8], *vtmp[8];
    int flags[4], out[8];


    /* Counters */
    int i, j;

    /* Set up the text and clipping rectangle */
    set_clip(bmp, 0, 0, bmp->w, bmp->h);
    text_mode(-1);
```

```
/* Initialize our vertex arrays */
for(i = 0; i < 4; i++)
{
    v[i] = &_v[i];
}

for(i = 0; i < 8; i++)
{
    vout[i] = &_vout[i];
    vtmp[i] = &_vtmp[i];
}

/* Set up four vertices with the world-space position of the box */
/* All of these boxes currently face due north */
v[0]->x = -BOXSIZE/2;
v[0]->y = -BOXSIZE/2;
v[0]->z = 0;

v[1]->x = -BOXSIZE/2;
v[1]->y = BOXSIZE/2;
v[1]->z = 0;

v[2]->x = BOXSIZE/2;
v[2]->y = BOXSIZE/2;
v[2]->z = 0;

v[3]->x = BOXSIZE/2;
v[3]->y = -BOXSIZE/2;
v[3]->z = 0;

/* Apply the camera & rotation matrix, translating worldspace to view space */
for(i = 0; i < 4; i++)
{
    apply_matrix_f(rotation, v[i]->x, v[i]->y, v[i]->z, &v[i]->x, &v[i]->y,
        &v[i]->z);
    apply_matrix_f(camera, v[i]->x, v[i]->y, v[i]->z, &v[i]->x, &v[i]->y,
        &v[i]->z);
    flags[i] = 0;

    /* set flags if the vertex is off of the edge of the screen */
    if (v[i]->x < -v[i]->z)
        flags[i] |= 1;
    else if (v[i]->x > v[i]->z)
        flags[i] |= 2;

    if (v[i]->y < -v[i]->z)
        flags[i] |= 4;
    else if (v[i]->y > v[i]->z)
        flags[i] |= 8;

    if (v[i]->z < 0.1)
        flags[i] |= 16;
}

/* Quit if all of the vertices are off of the edge of the screen */
if(flags[0] & flags[1] & flags[2] & flags[3])
{
    return;
}

/* Clip anything that is off of the edge of the screen */
if(flags[0] | flags[1] | flags[2] | flags[3])
{
    j = clip3d_f(POLYTYPE_FLAT, 0.1, 0.1, 4, (AL_CONST V3D_f **)v, vout,
        vtmp, out);
    if(j <= 0)
```

```
            return;
    }
    /*If nothing needed to be clipped, then draw stuff */
    else
    {
        vout[0] = v[0];
        vout[1] = v[1];
        vout[2] = v[2];
        vout[3] = v[3];

        j = 4;
    }

    /* project viewspace to screen space */
    for(i = 0; i < j; i++)
    {
        persp_project_f(vout[i]->x, vout[i]->y, vout[i]->z, &vout[i]->x,
            &vout[i]->y);
    }

    /* Draw the box */
    /* We draw a line from each corner to the next, cycling around to the first */
    for(i = 0; i < j; i++)
    {
            line(bmp, (int)vout[i]->x, (int)vout[i]->y, (int)vout[(i+1)%j]->x,
                (int)vout[(i+1)%j]->y, makecol(WHITE));
    }

    return;
}

/*-------------------------------------------------------------------------------
 * gfxDrawCircles
 *
 * Function Description:
 * This function draws a circle (sphere?) over every waypoint.
 *
 * Function Inputs:
 * *bmp - pointer to a bitmap to draw all of this stuff to.
 * *camera - a transformation matrix that contains information about the camera's
 * viewing angle
 * wp - Which waypoint is to be drawn
 *
 * Function Return:
 * None
 *
 *-------------------------------------------------------------------------------
 */

int gfxDrawCircles(BITMAP* bmp, MATRIX_f *camera, int wp)
{

    /* Arrays to hold our vertexes */
    V3D_f _v[2], _vout[4], _vtmp[4];
    V3D_f *v[2], *vout[4], *vtmp[4];
    int flags[2], out[4];
    double distance;

    /* Counters */
    int i, j;

    /* Set up the text and clippng rectangle */
    set_clip(bmp, 0, 0, bmp->w, bmp->h);
    text_mode(-1);

    /* Initialize our vertex arrays */
    for(i = 0; i < 2; i++)
```

```
{
    v[i] = &_v[i];
}

for(i = 0; i < 4; i++)
{
    vout[i] = &_vout[i];
    vtmp[i] = &_vtmp[i];
}

/* Set the circle's location */
v[0]->x = wayPoints.XYZ[wp].x;
v[0]->y = wayPoints.XYZ[wp].y + BOXSIZE/4;
v[0]->z = wayPoints.XYZ[wp].z;

v[1]->x = wayPoints.XYZ[wp].x;
v[1]->y = wayPoints.XYZ[wp].y - BOXSIZE/4;
v[1]->z = wayPoints.XYZ[wp].z;


/* Apply the camera matrix, translating world-space to view space */
for(i = 0; i < 2; i++)
{

    apply_matrix_f(camera, v[i]->x, v[i]->y, v[i]->z, &v[i]->x, &v[i]->y,
        &v[i]->z);
    flags[i] = 0;

    /* set flags if the vertex is off of the edge of the screen */
    if (v[i]->x < -v[i]->z)
        flags[i] |= 1;
    else if (v[i]->x > v[i]->z)
        flags[i] |= 2;

    if (v[i]->y < -v[i]->z)
        flags[i] |= 4;
    else if (v[i]->y > v[i]->z)
        flags[i] |= 8;

    if (v[i]->z < 0.1)
        flags[i] |= 16;
}

/* Quit if all of the vertices are off of the edge of the screen */
if(flags[0] & flags[1] & flags[2] & flags[3])
{

    return YES;
}

/* Clip anything that is off of the edge of the screen */
if(flags[0] | flags[1] | flags[2] | flags[3])
{
    j = clip3d_f(POLYTYPE_FLAT, 0.1, 0.1, 2, (AL_CONST V3D_f **)v, vout,
        vtmp, out);
    if(j <= 0)
        return YES;
}
/*If nothing needed to be clipped, then draw stuff */
else
{
    vout[0] = v[0];
    vout[1] = v[1];
    j = 2;
}

/* project viewspace to screen space */
```

```
    for(i = 0; i < j; i++)
    {
        persp_project_f(vout[i]->x, vout[i]->y, vout[i]->z, &vout[i]->x,
            &vout[i]->y);
    }


    /* Calculate the distance between the two vertexes and use that as the */
    /* circle diameter */
    distance = sqrt(((vout[0]->x - vout[1]->x) * (vout[0]->x - vout[1]->x)) +
        ((vout[0]->y - vout[1]->y) * (vout[0]->y - vout[1]->y)));

    /* Draw the visited waypoints red, and the non-visited waypoints green */
    if(wayPoints.visited[wp] == YES)
    {
        circlefill(bmp, (int)vout[0]->x, (int)(vout[0]->y - distance/2),
            (int)distance/2, makecol(RED));
    }
    else
    {
        circlefill(bmp, (int)vout[0]->x, (int)(vout[0]->y - distance/2),
            (int)distance/2, makecol(GREEN));
    }

    return NO;
}

/*-------------------------------------------------------------------------------
 * gfxDrawHeadingBody
 *
 * Function Description:
 * This function draws the heading scale at the top of the screen
 *
 * Function Inputs:
 * *bmp - pointer to a bitmap to draw all of this stuff to.
 *
 * Function Return:
 * None
 *
 *-------------------------------------------------------------------------------
 */
void gfxDrawHeadingBody(BITMAP *bmp)
{
    char buffer[10];
    int scale, middle, i;
    int headingArray[68];

    int bearing;

       /* Get bearing to the next waypoint */
    bearing = (int)infoGetBearingNextWaypoint();
    bearing = (int)headingBody - bearing;
    if(bearing < 0)
       bearing = bearing + 360;



    /* Find our heading in degrees */
    middle = (int)(headingBody * 180/PI);

    for (i = 0; i < 68; i++)
    {
        /* We need to scale our heading tape to the width of our screen */
        scale = i * (int)ceil((SCREEN_W/fov));
        headingArray[i] = (middle - ((int)(fov/2) - i)) % 360;

        /* Draw a vertical line every 15 degrees */
```

```
        if(headingArray[i] % 15 == 0)
        {
            vline(bmp, scale, 0, 7, makecol(WHITE));
            vline(bmp, scale-1, 0, 8, makecol(BLACK));
            vline(bmp, scale+1, 0, 8, makecol(BLACK));
            hline(bmp, scale-1, 8, scale+1, makecol(BLACK));
            /* Write the heading - this is temporary, I intend to do vector based */
            /* letters if time permits */
            sprintf(buffer, "%d", headingArray[i]);
            textout_centre(bmp, font, buffer, scale, 10, makecol(WHITE));

        }

        /* Draw a small line every 5 degrees */
        if(headingArray[i] % 5 == 0)
        {
            vline(bmp, scale, 0, 3, makecol(WHITE));
            vline(bmp, scale-1, 0, 4, makecol(BLACK));
            vline(bmp, scale+1, 0, 4, makecol(BLACK));
        }


        /* Draw arrow pointing to next waypoint */
        /* Can't do this yet, since we aren't reading in lat/long data */
        if(headingArray[i] == bearing)
        {
            line(bmp, scale, 14, scale-4, 20, makecol(WHITE));
            line(bmp, scale, 14, scale+4, 20, makecol(WHITE));
        }

     //  printf("middle = %d, headingArray[%d] = %d\n", middle, i, headingArray[i]);

    }

    /* If the pointer is too far to the right */
    if(bearing > headingArray[67])
    {
        line(bmp, SCREEN_W - 6, 0, SCREEN_W, 4, makecol(GREEN));
        line(bmp, SCREEN_W, 4, SCREEN_W - 6 , 8, makecol(GREEN));
    }

    /* If the pointer is too far to the left */
    else if(bearing < headingArray[0])
    {
        line(bmp, 0, 4, 6, 0, makecol(GREEN));
        line(bmp, 0, 4, 6, 8, makecol(GREEN));
    }




    return;
}


/*---------------------------------------------------------------------------
 * gfxRender
 *
 * Function Description:
 * This is the master drawing function.  It is called to render a complete frame.
 *
 * Function Inputs:
 * *bmp - pointer to a bitmap to draw all of this stuff to.
 *
 * Function Return:
 * None
 *
```

```
 *------------------------------------------------------------------------------
 */
void gfxRender (BITMAP *bmp)
{
    /* Camera related things */
    char buffer[80];                                    //Buffer for printing text
    MATRIX_f roller, camera, yrotation;                  //Camera orientation
    int x, y, w, h, i;                                  //Screen size info
    float xup, yup, zup;                                //Up vector info
    double ay;                                          //Angle to rotate boxes

       int bearing;

    /* Path calculating things */
    double distance = 0.0;                              //Distance to LZ
    double numBoxes = 0.0;                              //Number of boxes between
                                                        //two waypoints
    double dX = 0.0, dY = 0.0, dZ = 0.0;                //Distance between boxes
    double deltaX = 0.0, deltaY = 0.0, deltaZ = 0.0;
    double farWPX = 0.0, farWPY = 0.0, farWPZ = 0.0;    //Waypoint location
    double nearWPX = 0.0, nearWPY = 0.0, nearWPZ = 0.0; //Waypoint location
    double boxX = 0.0, boxY = 0.0, boxZ = 0.0;          //Box center location
    int farWP = 0, nearWP = 1;                          //Which waypoints we are
                                                        //navigating to

    int clipped = NO;                                   //Has a wp been clipped?

    /* Reset the bitmap */
    clear_to_color(bmp, makecol(BLACK));

    /* set up the viewport region */
    x = (SCREEN_W - viewport_w) / 2;
    y = (SCREEN_H - viewport_h) / 2;
    w = viewport_w;
    h = viewport_h;

    set_projection_viewport(x, y, w, h);
    rect(bmp, x-1, y-1, x+w, y+h, makecol(255, 0, 0));
    set_clip(bmp, x, y, x+w-1, y+h-1);

    /* Calculate the in-front vector */
    xfront = sin(headingBody) * cos(pitch);
    yfront = sin(pitch);
    zfront = cos(headingBody) * cos(pitch);

    /* Rotate the up vector around the in-front vector by the roll angle */
    get_vector_rotation_matrix_f(&roller, xfront, yfront, zfront, roll*128.0/PI);
    apply_matrix_f(&roller, 0, -1, 0, &xup, &yup, &zup);

    /* Build the camera matrix */
    get_camera_matrix_f(&camera,
                        xpos, ypos, zpos,       /* camera position */
                        xfront, yfront, zfront, /* in-front vector */
                        xup, yup, zup,          /* the up vector   */
                        fov, aspect);           /* camera settings */

    /* Add some text to the screen*/
    /* Most of this is here for debugging purposes  */
    /* Not really needed for the test version       */

    bearing = infoGetBearingNextWaypoint() %360;


//  set_clip(bmp, 0, 0, bmp->w, bmp->h);
//  text_mode(-1);
//  sprintf(buffer, "X position: %.2f", xpos);
//  textout(bmp, font, buffer, 0, 0, makecol(WHITE));
```

```
//  sprintf(buffer, "Y position: %.2f", -ypos);
//  textout(bmp, font, buffer, 0, 8, makecol(WHITE));

//  sprintf(buffer, "Z position: %.2f", zpos);
//  textout(bmp, font, buffer, 0, 16, makecol(WHITE));

    sprintf(buffer, "%d", (int)(headingBody*(180.0/PI)));
    textout_centre(bmp, font, buffer, SCREEN_W/2, 20, makecol(WHITE));


    /* Cycle through all of the waypoints.  Set the one closer to the LZ */
    /* as the far waypoint, and the one higher as the close waypoint.    */
    /* Determine the distance between the two waypoints, and how many    */
    /* boxes are there (right now, there are BOXDISTANCE meters between  */
    /* each box.  Find how far to rotate each box so that it is 90 deg   */
    /* to the path, then figure out its XYZ coordinates.  Draw the box   */
    /* and continue to the next set of waypoints.                        */

    while (farWP < wayPointCount-1)
    {
        farWPX = wayPoints.XYZ[farWP].x;
        farWPY = wayPoints.XYZ[farWP].y;
        farWPZ = wayPoints.XYZ[farWP].z;

        nearWPX = wayPoints.XYZ[nearWP].x;
        nearWPY = wayPoints.XYZ[nearWP].y;
        nearWPZ = wayPoints.XYZ[nearWP].z;

        dX = nearWPX - farWPX;
        dY = nearWPY - farWPY;
        dZ = nearWPZ - farWPZ;

        distance = sqrt((dX*dX) + (dY * dY) + (dZ*dZ));

        numBoxes = distance/BOXDISTANCE;

        if(numBoxes != 0.0)
        {
            deltaX = dX / numBoxes;
            deltaY = dY / numBoxes;
            deltaZ = dZ / numBoxes;
        }


        ay = DEG(atan2(dZ,dX));
//      ax = DEG(atan2(dY,dZ));


        clipped = gfxDrawCircles(bmp, &camera, farWP);

        boxX = farWPX;
        boxY = farWPY;
        boxZ = farWPZ;


        /* Compute the box's locations and draw them onto the bitmap */
        for(i = 0; i < numBoxes-1; i++)
        {
            boxX = boxX + deltaX;
            boxY = boxY + deltaY;
            boxZ = boxZ + deltaZ;

        //  printf("BoxX: %f\tboxY: %f\tBoxZ: %f\n", boxX, boxY, boxZ);

            dX = boxX - wayPoints.XYZ[nearWP].x;
            dY = boxX - wayPoints.XYZ[nearWP].y;
```

```
            dZ = boxX - wayPoints.XYZ[nearWP].z;

            if(((dX*dX) + (dY * dY) + (dZ*dZ)) > 10000.0)
            {
                get_transformation_matrix_f(&yrotation, 1, 0.0, ay + 64.0, 0.0,
                    boxX, boxY, boxZ);
                gfxDrawBoxes(bmp, &camera, &yrotation);
            }

        }


        farWP++;
        nearWP++;
    }

    /* If the waypoint is clipped, then draw the arrows guiding back to path */
    if(clipped == YES)
    {
//      gfxDrawArrows(bmp, &camera);
    }

    /* Draw any other things that need to be displayed */
    gfxDrawAltitude(bmp);
//  gfxDrawHeadingView(bmp);
    gfxDrawHeadingBody(bmp);

    return;
}

/*-----------------------------------------------------------------------------------------
 * gfxVerify
 *
 * Function Description:
 * This program prints a 'gods eye view' of the route, with north up.  It is not called
 * in the code, but is left in for legacy and debugging purposes.
 *
 * Function Inputs:
 * *bmp - pointer to a bitmap to draw all of this stuff to.
 *
 * Function Return:
 * success - whether or not we did this successfully.
 *
 *-----------------------------------------------------------------------------------------
 */
int gfxVerify(BITMAP *bmp)
{
       int success = FALSE, i;

       double normalized[NUMWP][3];

       double maxX = 0.0, maxZ = 0.0, minX = 0.0, minZ = 0.0;
       double deltaX, deltaZ;
       double scalefactorX, scalefactorZ;
       double largescale = 0.0;
       char center = 'A';
//     char buffer[80];

       /* Initialize the normalized array */
       for(i = 0; i < wayPointCount; i++)
       {
              normalized[i][X] = wayPoints.XYZ[i].x;
              normalized[i][Z] = wayPoints.XYZ[i].z;
       }

       /* Scale every waypoint to within screen size */
       /* Find the maximum and minimum X and Y values */
```

```
        for (i = 0; i < wayPointCount; i++)
        {
                if(wayPoints.XYZ[i].x > maxX)
                        maxX = wayPoints.XYZ[i].x;
                if(wayPoints.XYZ[i].x < minX)
                        minX = wayPoints.XYZ[i].x;
                if(wayPoints.XYZ[i].z > maxZ)
                        maxZ = wayPoints.XYZ[i].z;
                if(wayPoints.XYZ[i].z < minZ)
                        minZ = wayPoints.XYZ[i].z;
        }


        /* We want everything to be shifted up so that 0 is the minimum value in
           either normalized[i][0] or normalized[i][1] */
        for (i = 0; i < wayPointCount; i++)
        {
                normalized[i][X] += fabs(minX);
                normalized[i][Z] += fabs(minZ);
        }

        maxX = maxX + fabs(minX);
        maxZ = maxZ + fabs(minZ);

        /* Calculate the scaling factor - how much we shrink everything down so that all
           X and Z values are between 0 and SCREEN_W and SCREEN_H */
        scalefactorX = maxX / (SCREEN_W - 20);
        scalefactorZ = maxZ / (SCREEN_H - 20);

        /* When we shrink everything down, select the larger of the two scaling factors.
This will
           make everything relative to that scaling factor.  We then have to center
everything either
           horizontally or vertically */
        if(scalefactorX > scalefactorZ)
        {
                largescale = scalefactorX;
                center = 'H';
        }
        else
        {
                largescale = scalefactorZ;
                center = 'W';
        }

        /* Scale everything down by the normalization value */
        for (i = 0; i < wayPointCount; i++)
        {
                normalized[i][X] = normalized[i][X] / largescale;
                /* The SCREEN_H subtraction here is necessary to flip everything 'right
side up' */
                normalized[i][Z] = SCREEN_H - (normalized[i][Z] / largescale) - 5;
        }


        /* Center everything */
        minX = SCREEN_W;
        minZ = SCREEN_H;
        maxX = 0;
        maxZ = 0;
        /* First, center everything vertically */
        if (center == 'W')
        {
//              printf("Center = W!\n");
                for(i = 0; i < wayPointCount; i++)
                {
                        if(normalized[i][X] > maxX)
```

```
                        maxX = normalized[i][X];
                if(normalized[i][X] < minX)
                        minX = normalized[i][X];
        }

        deltaX = maxX - minX;

        deltaX = ((SCREEN_W - deltaX) / 2);

        for(i = 0; i < wayPointCount; i++)
        {
                normalized[i][X] += deltaX;
        }
}

/* Now center everything horizontally */
if(center == 'H')
{
//              printf("Center = H!\n");
        for(i = 0; i < wayPointCount; i++)
        {
                if(normalized[i][Z] > maxZ)
                        maxZ = normalized[i][Z];
                if(normalized[i][Z] < minZ)
                        minZ = normalized[i][Z];
        }

        deltaZ = maxZ - minZ;

        deltaZ = ((SCREEN_H - deltaZ) / 2);

        for(i = 0; i < wayPointCount; i++)
        {
                normalized[i][Z] += deltaZ;
        }
}


/* Finally, draw a circle around every waypoint, and draw lines between each
waypoint */
for (i = 0; i < wayPointCount; i++)
{
        circle(bmp, (int)normalized[i][X], (int)normalized[i][Z], 3,
makecol(WHITE));
        if (i < wayPointCount - 1)
                line(bmp, (int)normalized[i][X], (int)normalized[i][Z],
(int)normalized[i+1][X],
                        (int)normalized[i+1][Z], makecol(WHITE));
}


/* Draw a "North" arrow in the upper right corner */

line(bmp, (SCREEN_W - 6), 0, (SCREEN_W - 1), 5, makecol(WHITE));          /*  \
*/
line(bmp, (SCREEN_W - 6), 0, (SCREEN_W - 11), 5, makecol(WHITE));         /*  /
*/
line(bmp, (SCREEN_W - 6), 0, (SCREEN_W - 6), 11, makecol(WHITE));         /*  |
*/

/* Draw a N */
line(bmp, (SCREEN_W - 11), 21, (SCREEN_W - 11), 13, makecol(WHITE));
line(bmp, (SCREEN_W - 11), 13, (SCREEN_W - 1), 21, makecol(WHITE));
line(bmp, (SCREEN_W - 1), 21, (SCREEN_W - 1), 13, makecol(WHITE));


while(!key[KEY_Y])
```

```
        {
                blit(bmp, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);
        }
        return success;

}
/*-------------------------------------------------------------------------------
 * CalcDistance
 *
 * Function Description:
 * This function exists to calculate the XYZ coordinates of the waypoints relative
 * to the LZ. The LZ is at 0,0,0.  X is east/west (east positive), Y is up down
 * (up is negative), and Z is north south (north positive).  This is due to
 * allegro's coordinate system.
 *
 * Function Inputs:
 * None
 *
 * Function Return:
 * None
 *
 *-------------------------------------------------------------------------------
 */
void distCalcDistance(void)
{
    int i = 0;

    for (i = 0; i < NUMWP; i++)
    {
        wayPoints.XYZ[i].x = distXDistance(wayPoints.LLA[i][LAT],
            wayPoints.LLA[i][LON], wayPoints.LLA[0][LON]);

        wayPoints.XYZ[i].y = distYDistance(wayPoints.LLA[i][ALT], 0);

        wayPoints.XYZ[i].z = distZDistance(wayPoints.LLA[i][LAT],
                             wayPoints.LLA[0][LAT]);
    }
    return;
}

/*-------------------------------------------------------------------------------
 * XDistance
 *
 * Function Description:
 * This function calculates the east/west distance between two points.  If it
 * returns a positive, the first point is to the east of the second point.
 *
 * Function Inputs:
 * latA - The first point's latitude
 * lonA - The first point's longitude
 * lonB - The second point's longitude.  The second point will usually be the LZ.
 *
 * Function Return:
 * XDistance - the east/west distance, in meters, between the 1st and 2nd points.
 *
 *-------------------------------------------------------------------------------
 */
double distXDistance (double latA, double lonA, double lonB)
{
    double xDistance = 0.0, AOB = 0.0;

    /* This formula comes from the web */
    /* http://mathforum.org/library/drmath/view/51722.html */
    AOB = (cos(latA*PI/180) * cos(latA*PI/180) * cos((lonB - lonA)*PI/180)) +
        (sin(latA*PI/180) * sin(latA*PI/180));

    xDistance = acos(AOB) * RADIUS;
```

```
    if(lonA - lonB < 0.0)
    {
        xDistance = 0.0 - xDistance;
    }

    return xDistance;

}
/*-------------------------------------------------------------------------------
 * YDistance
 *
 * Function Description:
 * This function calculates the up/down distance between two points.  If it returns
 * a negative value, the first point is above the second point.
 *
 * Function Inputs:
 * altA - the altitude of the first point
 * altB - the altitude of the second point (usually the LZ)
 *
 * Function Return:
 * YDistance - the difference in altitude, in meters, between the two points.
 *-------------------------------------------------------------------------------
 */
double distYDistance(double altA, double altB)
{
    double yDistance = 0.0;

    yDistance = altA - altB;

    return yDistance;
}
/*-------------------------------------------------------------------------------
 * ZDistance
 *
 * Function Description:
 * This function calculates the north/south distance between two points.  If it
 * returns a positive value, the first point is north of the second point.
 *
 * Function Inputs:
 * latA - the latitude of the first point
 * latB - the latitude of the second point
 *
 * Function Return:
 * ZDistance - the north/south distance, in meters, between the two points.
 *
 *-------------------------------------------------------------------------------
 */

double distZDistance(double latA, double latB)
{
    double zDistance = 0.0, AOB = 0.0;

    /* Same formula as in distXDistance */
    /* Just substitute in different lat/longs */
    AOB = (cos(latA*PI/180) * cos(latB*PI/180)) + (sin(latA*PI/180) *
        sin(latB*PI/180));

    zDistance = acos(AOB) * RADIUS;

    if(latA - latB < 0.0)
    {
        zDistance = 0.0 - zDistance;
    }

    return zDistance;
}
```

```
/*-------------------------------------------------------------------------------
 * calcDistBetweenPoints
 * This function calculates the distance between two latitude and longitude pts.
 *-------------------------------------------------------------------------------
 */

double calcDistBetweenPoints(double lat1, double lon1, double lat2, double lon2)
{
	double distance, dX, dZ;

	dX = distXDistance(lat1, lon1, lon2);
	dZ = distZDistance(lat1, lat2);

	distance = (dX*dX) + (dZ*dZ);
	distance = sqrt(distance);

	return distance;
}


/*-------------------------------------------------------------------------------
 * calcHeading
 * This function calculates the user's heading based on his new and old position.
 *-------------------------------------------------------------------------------
 */

double calcHeading(void)
{

	double dX, dZ;
	double heading;

	dX = currentData.velocity[0];

	dZ = currentData.velocity[2];

	if(dX == 0 && dZ == 0)
	{
		heading = headingBody;
	}
	else
	{
		heading = atan2(dZ, dX);
		heading = fabs(heading - 1.5707963267948966);
		if(dX <= 0 && dZ >= 0)
		{
			heading = 6.28318530717958 - heading;
		}
	}

    return heading;
}

/*-------------------------------------------------------------------------------
 * timerLogicCounter
 * This function is the timer.  The program should run at BPM beats per second.
 *-------------------------------------------------------------------------------
 */
void timerLogicCounter(void)
{
    logicCounter++;

    return;
}

/* End of file */
```

Appendix B – Effects of the AN/PVS-21 NVGs On the MT-9 IMU

Introduction

The AN/PVS-21 night vision goggles (NVGs) have two power units that each generate about 3,000 volts mounted on the NVG body. These power units may cause electro-magnetic interference with other electronics. Of particular concern is any potential for interference with an MT-9 MEMs sensor, as this sensor will be used to track head movement in order to generate a Heads Up Display (HUDS). The MT-9 packages accelerometers, gyros, and magnetometers, each capable of detecting movement in any of three axes. The NVGs power units may affect any of these sensors, and therefore, this interference must be measured.

Experimental Setup

This problem was approached as a 2^kr full factorial experiment. There are two factors: the sensors in the MT-9, and the power draw of the NVGs. The first factor has nine levels – three accelerometer sensors, three gyro sensors, and three magnetometer sensors. The second factor has five levels – baseline with the NVGs far away from the MT-9; NVGs off but next to the MT-9; NVGs on with the lens covered so no light can enter; NVGs on in a darkened office room; and NVGs on with the office lights on. Two replications of the experiment were conducted.

Each variation in the NVGs situation is designed to test various possible levels of interference. The baseline is to determine what the MT-9 will read without any possible outside influence. Secondly, the NVGs will be powered down and placed next to the MT-9 to determine if the metal in the NVGs body will affect the MT-9. Next, the NVGs are turned on. The NVGs draw different amounts of power depending on how much

ambient light there is to amplify, drawing less power when there is less light. Thus, the NVGs are tested with the light-collection lens completely covered, in order to draw the least amount of power. Next, the lens are uncovered, and the lens allowed to collect the light of a darkened office room. Finally, the lights in the office are turned up, making the goggles draw as much power as possible.

A MATLAB program was used to interface with the MT-9 unit and collect data. Each experiment consisted of setting up the NVGs in accordance with the testing procedures, and using the MATLAB program to collect 30 seconds worth of data. This data was saved as a tab-delimited text file. Each sensor's reading is formatted as a unitless integer. These raw measurements do not directly translate to any familiar unit (for instance, m/s^2 for the accelerometers); further processing must be performed in order to do the conversion. At the moment, this conversion is not possible.

Analysis

First, the average value for each sensor in each experiment was calculated.

Table 3: Average values recorded for each sensor

	Accel 1	Accel 2	Accel3	Gyro 1	Gyro 2	Gyro 3	Mag 1	Mag 2	Mag 3
Baseline 1	32160.45	35476.14	32848.35	32831.35	32836.03	36411.72	30900.28	28408.16	6473.44
Baseline 2	32183.31	35395.96	32841.33	32836.70	32842.89	35618.40	31140.02	27459.27	6998.08
NVG Off 1	32179.72	35384.69	32844.37	32831.51	32839.33	35564.12	31054.09	27396.46	6623.95
NVG Off 2	32178.58	35398.67	32839.47	32838.00	32843.89	35750.80	31272.34	27386.04	7035.52
NVG On Covered 1	32179.31	35398.20	32858.26	32831.92	32840.00	37615.70	31873.77	27621.89	6782.29
NVG On Covered 2	32174.82	35399.76	32851.86	32834.73	32844.18	36529.46	32138.89	27531.21	7066.08
NVG On Dim 1	32181.47	35399.00	32836.11	32834.67	32842.90	35800.11	31100.52	27383.52	6860.37
NVG On Dim 2	32176.36	35400.44	32841.33	32836.29	32844.36	35769.25	31283.76	27385.24	7094.19
NVG On Bright 1	32181.91	35399.18	32838.05	32836.83	32843.03	35791.96	31100.95	27389.19	6930.45
NVG On Bright 2	32169.91	35446.62	32835.06	32837.06	32845.80	36433.82	31336.38	27723.50	7055.84

Once this was completed, an Analysis of Variance (ANOVA) was conducted on the experimental data:

Table 4: ANOVA conducted on the data obtained during the experiments

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
NVGs	876725.4002	4	219181.3501	4.7562843	0.0027548	2.5787392
Sensors	6369596255	8	796199531.9	17277.708	8.305E-76	2.1521329
Interaction	3331378.55	32	104105.5797	2.2591144	0.0059112	1.6997564
Within	2073711.354	45	46082.47453			
Total	6375878070	89				

With the information in the ANOVA, the standard deviation of the effects can be calculated and used to construct a 90% confidence interval (z=1.645) for the effects that the NVGs have upon the sensors, as seen in Table 5.

Table 5: Mean Effect, standard deviation, and confidence intervals for various states of the NVGs

Parameter	Mean Effect	Standard Deviation	Lower Bound of CI	Upper Bound of CI
Baseline	29758.99	45.26	29684.55	29833.44
NVG Off	29681.20	45.26	29606.75	29755.64
NVG On - Covered	29965.13	45.26	29890.68	30039.58
NVG On – Dim	29714.99	45.26	29640.55	29789.44
NVG On – Bright	29777.53	45.26	29703.08	29851.98

If the confidence intervals overlap, then there is 90% confidence that there is no difference between the two values. From this, it can be seen that there is no difference between the baseline measurements and when the NVGs are off, or on with dim or bright

light. However, there is a significant difference between the other readings and when the NVGs are on and covered. Figure 34 illustrates the confidence levels graphically:

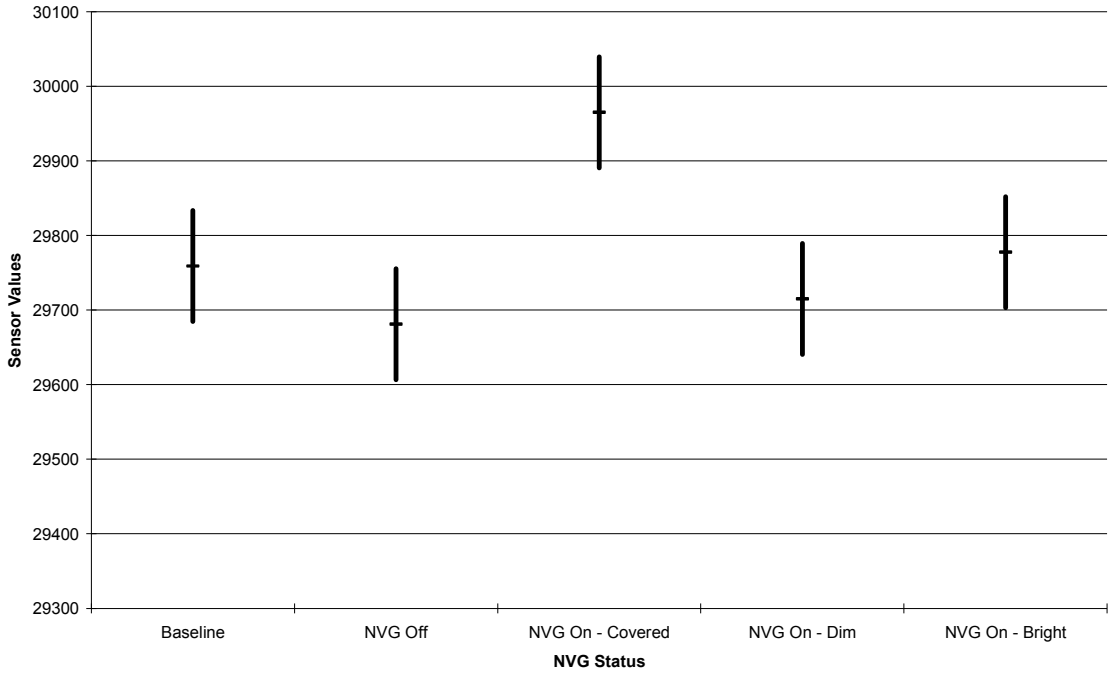


Figure 34: 90% Confidence Intervals for the Various NVG States

This indicates that, for some reason, if the goggles are turned on with no light entering them, they have an effect on the readings of the MT-9 sensors. This finding does not immediately make sense. The NVGs draw more power, and therefore should interfere more with the MT-9, when there is more light available, because it amplifies more light. When it is totally dark, the NVGs do not have to amplify any light at all, and should draw less power.

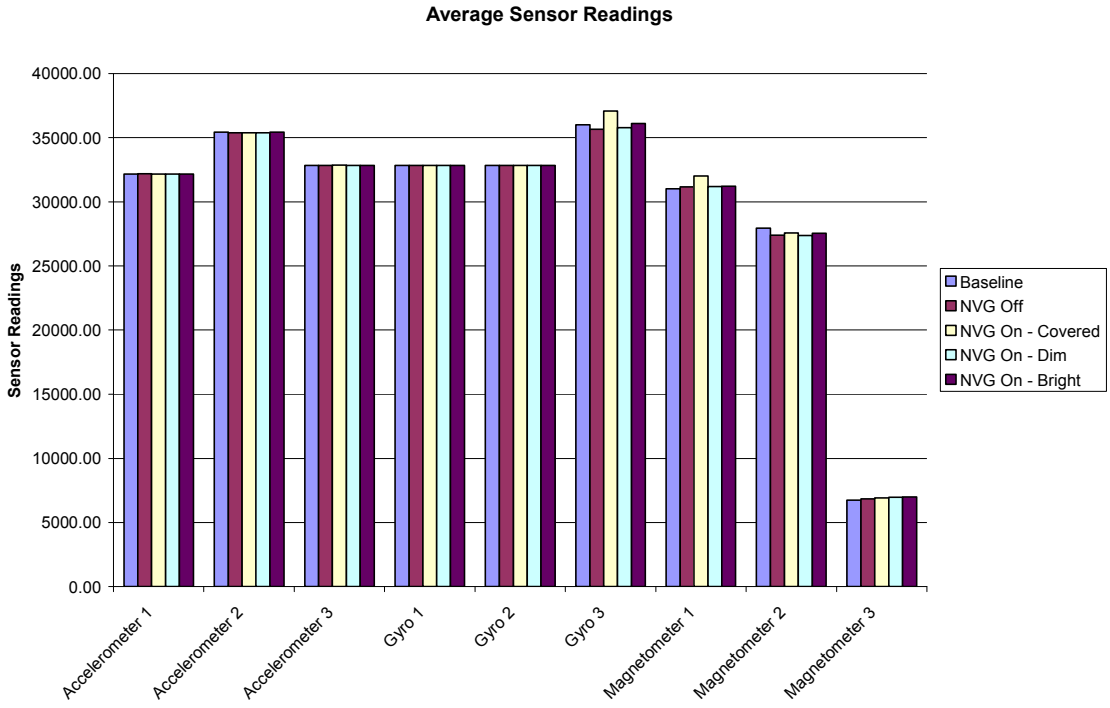


Figure 35: Average Sensor Readings For Various NVG States

Figure 35 depicts the average sensor readings taken during the experiments. As can be seen, in most instances, no matter what the NVGs situation, the sensors read the same. However, in some cases, there are discrepancies, mostly with the NVGs turned on but covered. The most significant differences in reading come from the Gyro 3 sensor and the Magnetometer 1 sensors. Figure 36 and Figure 37 show the difference in readings between the baseline and NVGs on but covered experiments for Gyroscope 1.

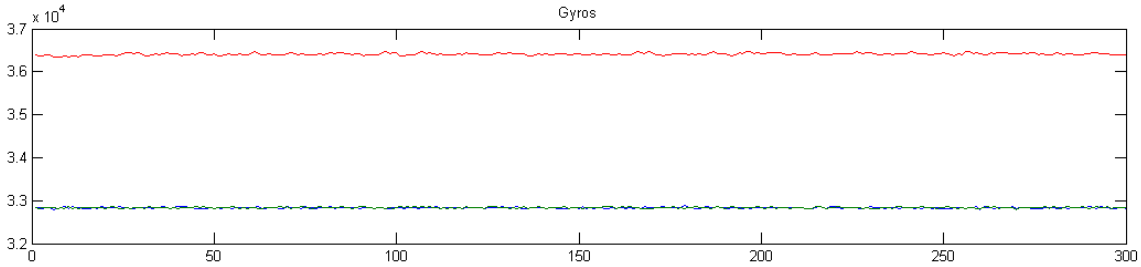


Figure 36: Gyroscope Measurements During The Baseline Experiment.

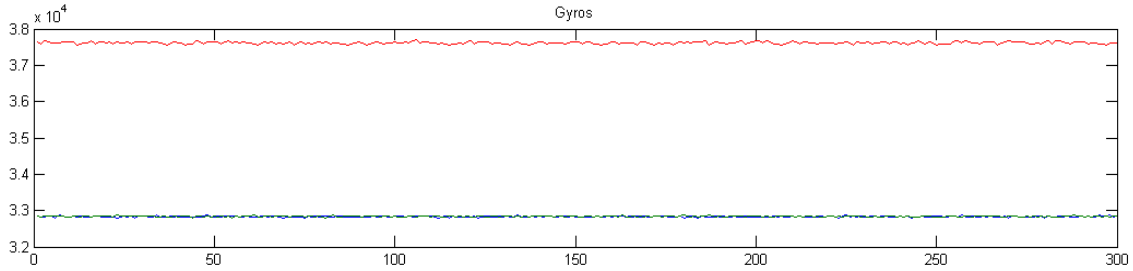


Figure 37: Gyroscope Measurements While The NVGS Were Turned On, But Covered.

Conclusion

The NVGs do not affect the MT-9's sensors, except for the case when the NVGs are turned on, but covered. This does not make any immediate sense, as this is the case when the NVGs are drawing the least amount of power. It is possible that the NVGS attempt to increase the gain in response to the lack of electrons present. This could result in increased leakage of energy, interfering with the MT-9's readings.

There are several other possible explanations for these results. The first, and most likely, is experimental error. The particular MT-9 unit used in the experiment is a rather old unit, and may not be entirely reliable. Most MEMS devices do not have a long service life, and this may introduce inaccuracies to the measurements. Further, the MT-9 is known to have a long 'warm-up' period, where it may return biased measurements until it has reached a steady state. Although the MT-9 was warmed up for a long period before measurements were taken, the unit may not have yet reached steady state.

It should also be noted that the NVGs only showed an impact on the MT-9 sensor when the NVGs were powered up and the lens completely covered. When the NVGs were operating in the dark room, the MT-9 behaved absolutely normally, returning measurements that were the same as the baseline measurements. This is the state that the NVGs will be in, in most cases.

Further testing may be necessary to gain further confidence in the results gained. However, at the moment, at the 90% confidence level, there is no impact on the MT-9 sensor when it is operating next to the NVGs in most instances.

Bibliography

[ACK94] Ackroyd, Neil, and Robert Lorimer. "Global Navigation: A GPS User's Guide." London: Lloyd's of London Press. 1994.

[BAL03] Balaz, Brian. "A Three-Dimensional Heads-Up Primary Navigation Reference Display For Paratroopers Performing High Altitude High Open Jumps." 2003.

[BRA04] Brain, Marshall, and Tom Harris. "How GPS Receivers Work." http://electronics.howstuffworks.com/gps.htm/printable. 2004.

[GAR05] Garmin Incorporated. "RINO 120 2-Way Radio and Personal Navigator Owner's Manual." Taiwain. 2005.

[GER03] Germain, Brian. "The Parachute and Its Pilot." 2004.

[HAR05] Hargreaves, Shawn. "A General Introduction to Allegro." http://www.talula.demon.co.uk/allegro/onlinedocs/en/readme.html 2005.

[IGE00] IGEB Executive Secretariat. "GPS & Selective Availability Q&A." 2000.

[JOF03] Joffrion, Jacque, and John Raquet. "Head Tracking for 3D Audio Using a GPS-Aided MEMS IMU." Air Force Insitute of Technology, 2003.

[KAM03] Kaminer, Isaac, and Oleg Yakimenko. "Development Of Control Algorithm For The Autonomous Gliding Delivery System." AIAA Journal AIAA-2003-2116. 2003.

[MIT97] Mittal, S. and T. Tezduyar. "Finite Element Simulation of Large Ram-Air Parachutes." Seminar Proceedings of National Symposium on Parachute and Lighter-than-Air Systems Technologies. Para India. 1997.

[MUL94] Mulder, Axel. "Human Movement Tracking Technology." Hand Centered Studies of Human Movement Project, Technical Report 94-1.

[NAV03] Navstar GPS Space Division. http://gps.losangeles.af.mil/space/index.htm 2003.

[NEW00] Newman, Richard. "HUDs, HMDs, and SDO: A Problem or a Bad Reputation." Recent Trends in Spatial Disorientation Conference. 2000.

[PAG95] Page, Scott, et al. "The Global Positioning System – Assessing National Policies." Rand Corporation. 1995.

[PAR03] Parrish, Russel V. "Avionic Pictorial Tunnel-/Pathway-/Highway-In-The-Sky Workshops." Langley Research Center. 2003.

[PRI03] Prinzel, Lawrence, et al. "Evaluation of Tunnel Concepts for Advanced Aviation Displays." NASA Langley Research Center. 2003.

[RAS98] Rash, Clarence, et al. "Helmet Mounted Displays: Design Issues for Rotary Aircraft." U.S. Army Aeromedical Research Laboratory. 1998.

[SOB94] Sobieski, Jerry. "The Aerodynamics and Piloting of High Performance Ram-Air Parachutes." 1994.

[SNO99] Snow, Michael, et al. "Flying Complex Approaches Using A Head-Up Display: Effects Of Visibility And Display Type." 1999.

[SSK05] SSK Industries, Inc. "Operational Paratroopers Navigational System for Paratroopers using High Altitude High Opening Parachutes." 2005.

[STS03a] Specialized Technical Services. "AFIT Thesis Topic Proposal from STS." 2003.

[STS03b] Specialized Technical Services. "HUD." 2003.

[STS03c] Specialized Technical Services. http://www.sts-eo.com 2003.

[WIT04] Witte, T.H., and A.M. Wilson. "Accuracy of non-differential GPS for the determination of speed over ground." Journal of Biomechanics. 2004.

[WIL05] Wilson, David. "David L. Wilson's GPS Accuracy Page." http://users.erols.com/dlwilson/gps.htm

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
21-03-2005	Master's Thesis	August 2003 – March 2005

3. TITLE AND SUBTITLE

A THREE DIMENSIONAL HELMET MOUNTED PRIMARY FLIGHT REFERENCE FOR PARATROOPERS

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Thompson, Jason I., Second Lieutenant, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Bldg 641
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/05-18

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Specialized Technical Services
2794 Indian Ripple Road
Beavercreek, Ohio 45440. USA
(937) 426-2341

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This thesis seeks to develop a Heads Up Display (HUD) presented on a Helmet Mounted Display (HMD), which presents a three-dimensional, graphical, predictive navigational reference to a paratrooper during a High Altitude, High Opening (HAHO) parachute jump. A Path Generating Algorithm (PGA) takes as input the Landing Zone's (LZ) location, the wind profile, and the paratrooper's parachute's performance characteristics, and returns a set of waypoints for the paratrooper to follow. The PGA attempts to maximize the distance that the paratrooper travels.

The PGA's output is used to build a path to the LZ from a Release Point (RP). During the jump, GPS signals and an Inertial Measurement Unit functioning as a head tracker is used to determine the user's location and head orientation. The HUD presents a virtual "Tunnel in the Sky" for the paratrooper to navigate through.

Due to hardware unavailability, the head tracker could not be implemented. Ground testing of the system sans the head tracker determined that subjects using the graphical HUD navigated a path slower than when compared to a baseline navigational reference. It is theorized that implementing the head tracker will make the HUD more efficient and better suited to navigation.

15. SUBJECT TERMS

NAVIGATION REFERENCE, PARACHUTE JUMPING, THESES, THREE DIMENSIONAL, GLOBAL POSITIONING SYSTEM, HIGH ALTITUDE, COMPUTER GRAPHICS, NIGHT VISION DEVICES, HEAD UP DISPLAYS, PARACHUTISTS, PARATROOPERS.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			John F. Raquet, Ph.D.
U	U	U	UU	145	19b. TELEPHONE NUMBER *(Include area code)* (937) 785-3636 x4580 (John.Raquet@afit.edu)