A Mapping System for an Autonomous Helicopter

by

Russell Sammon

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical [Computer] Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

© Russell P. Sammon, 1999. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author: .....
Department of Electrical Engineering and Computer Science
May 18, 1999

Certified by: .....
Paul A. DeBitetto
Senior Member of Technical Staff, Charles Stark Draper Laboratory
Thesis Supervisor

Certified by: .....
Prof. Seth Teller
Associate Professor of Computer Science and Engineering, EECS Department,
Massachusetts Institute of Technology
Thesis Supervisor

Accepted by: .....
Prof. Arthur C. Smith
Chairman, Department Committee on Graduate Theses

ENG
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 15 1999
LIBRARIES

Acknowledgments

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Internal Research & Development No. 18598.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

The work that is described in this document was by no means done all by myself. There are many people that I would like to thank for their support of my education here at MIT and Draper Laboratory during the past few years.

I would like to thank the members of the helicopter team, who have greatly influenced my development as an engineer: Paul Debitetto, Christian Trott, Bob Butler, Long Phan, Mike Piedmonte, and Anthony Lorusso. Special thanks goes to Paul Debitetto for his thoughtful critiquing and leadership. Long Phan was my close partner in the design of the scanning laser rangefinder, and this thesis would not have been possible without his inspiration and expertise.

During my work on the mapping system, a number of Draper staff took time out of their own busy schedules to help me. I am especially indebted to Chris Sanders, Chris Smith, John Plump, Linda Leonard, John Danis, and Dave Hauger for their patience and helpfulness over the last year.

I would also like to thank my MIT advisor, Seth Teller, for helping me to graduate this year.

Other Draper fellows and students in the autonomous vehicle lab have also provided me with both technical expertise and encouragement. I would like to thank Mohan Gurunathan, Jonah Peskin, and Bill Kaliardos for their advice and moral support.

Finally, I would like to thank my entire family for years of encouragement and support. It is to my mother, father, and sister that I dedicate this thesis.

Rusty Sammon May, 1999

Contents

List of Figures

Chapter 1

Introduction

1.1 The Draper Autonomous Helicopter

Some day the United States military hopes to have a small aerial robot that can function entirely autonomously. If their wish comes true, it would be possible for a single soldier to deploy many of these robots and assign them various different tasks. One might be ordered to fly into a small town a few miles away and verify the location of various strategic buildings and structures in this town. Two others might be assigned to search cooperatively the surrounding countryside for enemy vehicles and report back to home base if they find anything. Still another might be instructed to fly into enemy territory, locate targets, and plant bombs where necessary, all the while keeping a lookout for wounded American soldiers.

These are ambitious goals, but not impossible ones. Every year, more precise sensors are developed, faster computers are built, and engineers develop innovative new solutions to the challenges of autonomous flight. The Draper Small Autonomous Aerial Vehicle (DSAAV) project is an ongoing effort by Draper Laboratories to produce a small, fully autonomous helicopter suitable for military applications. It is hoped that eventually the helicopter will be able to complete entire missions on its own, though the current helicopter does little more than maintain flight.

The current DSAAV is a modified Bergen radio-controlled helicopter. In addition to the normal electronics required for controlling a radio-controlled helicopter, the Bergen also carries a special box containing all the electronics necessary for autonomous flight. This box contains an electronic power supply, various sensors, an on-board 486-compatible computer system, and a radio modem that is used to communicate with the helicopter ground station. The ground station is a laptop computer system that allows the helicopter operator to enter commands, observe behavior, and log data from the helicopter.

Maintaining autonomous flight is not a simple task. The current DSAAV uses a global positioning system (GPS), inertial measurement unit (IMU), sonar altimeter, and electronic compass to determine its own location and orientation. The measurements made by these sensors are combined using a Kalman filter to produce best estimates of the helicopter's current position, orientation, and motion. The on-board computer system uses these estimates to maintain autonomous flight and navigate the helicopter along flight paths specified from the ground station.

1.2 The Need for Obstacle Detection

One feature that is distinctly absent from the current helicopter is the ability to detect obstacles in its environment. Presently, the only obstacle avoidance sensor of any sort on the helicopter is the sonar altimeter. While the sonar altimeter is very useful for takeoff and landing, it points downwards and is therefore not at all useful for flight-path obstacle avoidance. Since the helicopter can not detect objects such as trees, buildings, and hill sides, there is no way for the helicopter to avoid these objects autonomously. Hence it is unwise to fly the current helicopter in unknown, obstacle-rich environments because of the high likelihood of a collision. At present, urban missions are out of the question, and the helicopter functionality is limited to flat, wide-open fields.

The addition of an obstacle detection sensor of some sort would greatly improve the versatility of the autonomous helicopter. The measurements from the ranging sensor could be used to create a map of obstacle locations, which would in turn be useful to the helicopter itself, soldiers, and other autonomous vehicles. Once the helicopter knows the locations of objects in its local environment, flight paths can be planned to avoid collisions. The helicopter then would be able fly into obstacle-rich areas such as urban and mountainous terrain and could be useful for surveillance purposes by detecting and locating specific objects. Obstacle detection is an important stepping stone to creating a fully autonomous vehicle.

1.3 The Scanning Laser Rangefinder

1.3.1 Objectives

The first objective of the scanning laser rangefinder project was to create a ranging sensor for the DSAAV which would allow the helicopter to perform high speed obstacle avoidance. The hope is that the helicopter will eventually be able to fly at 35 miles per hour down a city street and be able to avoid oncoming obstacles such as buildings

and automobiles. To do this requires a ranging sensor that has a long range, a fast sampling rate, medium to high precision, good noise rejection, and a wide field of view. Also, in order to avoid obstacles, the range data produced by the sensor must be processed in real time.

The secondary objective of the project was to produce a mapping system that would be useful for volumetric mapping, three-dimensional path planning, and landmark-based navigation. The helicopter should be able to fly into an unknown area and then identify and locate obstacles. The positions of these obstacles need to be saved for future use by this autonomous helicopter and other military units, including other autonomous vehicles. Finally, it is hoped that a high resolution map would allow the helicopter to determine its own position by noting the relative angle and distance of landmark objects.[1]

In addition, Draper Laboratories maintains a sophisticated, real-time simulation program that permits quick and safe testing of software on-board the helicopter and on the ground station. This simulation program is invaluable in developing new navigation algorithms and testing them under a variety of conditions. The addition of any new hardware or software to the helicopter is not complete until appropriate simulation code has been written. Therefore one final objective is to provide the necessary simulation foundation for future improvements to the DSAAV software.

It is important to keep in mind that the DSAAV is constantly changing. Software algorithms are updated continuously to improve the functionality of the helicopter. Hardware components are replaced when newer, better technology becomes available. This means all modifications to the DSAAV should be made keeping in mind the possible future improvements to the system. For example, it is foreseeable that some day the time-of-flight laser rangefinder will be replaced by another form of rangefinding sensor (such as radar or sonar). Therefore the new mapping system should be designed to be extensible and interchangeable.

1.3.2 Extent of Project

This thesis describes only the foundation for mapping system suitable for use with a laser scanner on the autonomous helicopter. It does not describe the autonomous helicopter itself in any detail, and gives only a basic synopsis of the laser scanner

[1] The helicopter's estimate of its own position is produced using a Kalman filter that combines measurements from the GPS, IMU, compass, and sonar altimeter. In the current implementation of the DSAAV, this position estimate is very dependent on the GPS. Should the GPS stop working for any reason (ex: poor weather prevents contact with GPS satellites), the Kalman filter position estimate is greatly degraded and highly susceptible to the inherent drift of the IMU. Checking the location of the helicopter relative to a known landmark will create another helicopter position measurement that could augment the Kalman filter estimate.

hardware. For a better description of the helicopter, the reader is directed to [23]. A complete description of the scanning laser rangefinder is given in [18].

The addition of obstacle detection hardware and software to the autonomous helicopter creates opportunities for a number of improvements to the DSAAV. This thesis does not attempt to implement all possible functionalities of the new sensor, but instead provides a foundation for future work on the DSAAV. It specifically does not include obstacle avoidance algorithms, object recognition, search patterns, or multiple helicopter coordination. Opportunities such as these for additional improvements to the DSAAV are described in Section 9.1.

1.3.3 Project Overview

Draper Fellow Long Phan and I have created a prototype system for the use of a scanning laser rangefinder sensor on the DSAAV. Long Phan built the hardware portion of the system, which consists of the pulsed laser, a servo-base scanning system, and a high speed timer. This thesis focuses on the software portion of the laser scanner implementation, including data communication and filtering, as well as the map structures and generation algorithms.

In order to design the software portion of mapping system, it is necessary to understand the basics of the hardware portion of the system. Chapter 2 describes the hardware portion of the scanning laser rangefinder. This assembly uses laser time-of-flight measurements to obtain distances from the helicopter to reflective objects in the field of view. The laser scans one-dimensionally in a horizontal pattern, making distance measurements at discrete angles relative to the helicopter. This hardware produces range and angle measurements that are sent to the helicopter's on-board computer for further processing.

Chapter 3 through 8 describe the implementation of the preliminary software necessary to make use of the laser scanner data. Chapter 3 describes the requirements for a mapping system suitable for an autonomous helicopter and outlines the need for a multi-level mapping system. Chapter 4 gives a brief overview of some mapping possible mapping representations that have been used successfully with other autonomous vehicles. Chapter 5 rationalizes the selection of the combination mapping system. Chapter 6 describes the first level of the mapping system, a certainty grid representation which is used for compilation of range measurements. Chapter 7 describes the second level of the mapping system, a kd-tree representation which provides efficient storage of the volumetric data compiled in the certainty grid. Chapter 8 gives an overview of the simulations implementation and the mapping system results.

Chapter 9 describes the simulation and testing of the mapping system software and characterizes it's suitability for use on the DSAAV. This chapter also notes some possible uses of the laser scanner and mapping system on the autonomous helicopter and offers insight into the future of the DSAAV.

Chapter 2

Hardware

This chapter describes the hardware portion of the scanning laser rangefinder system. While this hardware is not part of the mapping system, an understanding of the capabilities and limitations of the hardware is necessary for designing the software. The mapping system should be tailored to the range, resolution, sampling rate, data format, possible errors, and scan pattern of the laser rangefinder hardware, among other things. While it is likely that the hardware parameters will change in the future, the present system still provides a prototype that will be useful for the initial evaluation of the mapping system. The information provided in this chapter may also be useful as a reference for future work in implementing the scanning laser rangefinder on the DSAAV.

2.1 Overview

Long Phan[1] designed and built a custom scanning laser rangefinder for use on the Draper autonomous helicopter. The design and testing of this laser rangefinder served as Phan's Masters of Engineering thesis.[18] Using time of flight measurements of a pulsed laser, the scanning laser rangefinder produces precise measurements of the range to reflective objects in the field of view. The assembly consists of four main parts: the modulating laser itself, the precision timing module, the scanning mechanism, and a PIC microcontroller that controls the sensor sampling and communication.

The laser is pulsed at a rate of 500 Hz. The round-trip time of each laser pulse is measured by the timing module and is proportional to the distance traveled by the laser pulse. Together, the laser and timing module produce 500 range measurements every second. In order to obtain a wider field of view, the laser is scanned in a

[1] another graduate student working at Draper Labs.

horizontal sweep pattern at a rate of 2 sweeps per second. The current angle of the laser is recorded at the time of each laser pulse and then paired with the corresponding range measurement for transmission to the helicopter's on-board 486 computer. Coordination of this entire process is performed by the PIC microcontroller.

2.2 The Laser and Detector

2.2.1 Specifications

The laser itself is a 100 W pulsed laser. It has a wavelength of 905 nm, and an approximate fanout of 10°. It can be pulsed at a rate as high as 5 kHz. The time of flight counter is a IMRA RC1202 and is capable of determining time resolution to within 500 ps.

The laser beam can be modeled as a cone with tip angle of 10° (equivalent to the fanout of the laser beam). Objects that fall within this cone will reflect some portion of the laser light to the laser detector. The laser detector is a ERX1B which has a built-in comparator and an area of 1 mm^2. When the laser detector senses the return of a laser pulse, it sends a signal to stop the counter in the timing module.

Together, the pulsed laser and detector have a maximum range of 65 ft in overcast conditions, and 45 ft in bright sunlight. Bright sunlight decreases the maximum range of the laser by adding ambient light that can not be distinguished from the 905 nm laser wavelength. While the laser was chosen to have a frequency at which minimal ambient light was present, the effect of sunlight can not be ignored. The long range of the laser is made possible by the high power of the laser and the sensitive diode. It is speculated that with a more focused laser beam and a larger lens on the detector that the maximum range of the laser rangefinder could be increased by a factor of two or three.[18]

2.2.2 Safety

One important consideration in the choice of the laser was safety. Lasers can can cause permanent eye damage and blindness to people who are unfortunate enough to look into the beam. Any laser that is placed on an autonomous vehicle operating in an environment where there are humans and animals present must be eye-safe. While exact numbers are not available as to the incident power of the laser, it has been shown to be eye-safe.[18]

2.2.3 Visibility

The laser wavelength of 905 nm is invisible to the human eye. This is particularly important for military applications, where a visible laser could inform the enemy as to the presence and location of the autonomous helicopter.

2.3 The Timing Module

The round trip travel time of the laser pulse is measured by the IMRA RC1202 precision timing module. This timing module uses technology to achieve a timing resolution of 500 ps on a 12-bit digital counter. The timer is configured on startup by loading a data control word that specifies the resolution and trigger parameters. Configuration is performed by the PIC microcontroller, which is described in detail in Section 2.5.

The 500 ps resolution of the timing module gives the laser impressive range precision. The range precision can be calculated from the timing resolution using the speed of light as shown below in Equation 2.1. Note the factor of 1/2 that is included in the range precision formula to account for the fact that the laser pulse travels double the distance to the reflecting object (once as it goes out to the object, once as it comes back from the object).

$$range\ resolution = 1/2 * (timing\ resolution) * (speed\ of\ light) \qquad (2.1)$$

The range precision of approximately 2.2 inches/count allows for precise measurements of the locations of objects relative to the helicopter's position. The accuracy of these measurements has not yet been determined, though it is expected that the laser will produce some errors because of the aforementioned specular reflections and ambient light.

2.4 The Scanning Mechanism

In order to achieve a wider field of view and obtain range measurements for a wide area, the pulsed laser and detector assembly is scanned in a horizontal left-right fashion. In the current version of the laser, there is no vertical component to the scanning. Future versions of the scanning laser rangefinder may implement horizontal and vertical scanning (like a raster), or may gimbal the entire laser assembly control the viewing direction of the laser relative to the helicopter.

The scanning mechanism is configured to scan the laser from $\theta = -80°$ to $+80°$

($\theta = 0°$ is directly in front of the autonomous helicopter). The scanning motion is performed by a Futaba servo. The position of the servo is controlled by the duty cycle of a the servo's pulse width modulation (PWM) input. The servo's position is updated at a rate of approximately 50 Hz, with PWM duty cycles ranging from 0.4 ms to 2.4 ms. A duty cycle of 0.4 ms corresponds to the servo position of -80°, and a duty cycle of 2.4 ms corresponds to a servo position of +80°.

The pulse width modulation servo input is provided by a Basic STAMP microcontroller. The Basic STAMP was chosen for this job because if provides a quick, simple, and accurate means of performing pulse width modulation at low frequencies such as 50 Hz. Scanning is performed by alternating the duty cycle of the STAMP PWM output between 0.4 ms and 2.4 ms. This causes the servo arm to move from -80° to +80° and back again, repeating this cycle every second.

The position of the servo arm is measured using a built-in potentiometer on the Futaba servo. A voltage of +5 V is applied across the potentiometer to produce a reference voltage proportional to the servo position at the center tap. This reference voltage is sampled by the PIC microcontroller at the time of each laser pulse to find the current servo angle. This servo angle is then paired with the corresponding range from the timing module and sent to the autonomous helicopter's on-board 486 computer.

2.5 The PIC Microcontroller

2.5.1 Selection of PIC16C73A

A Microchip PIC16C73A microcontroller controls all the laser pulsing, timing mechanism configuration and triggering, range sampling, servo angle sampling, and communication with the on-board computer. The choice of the PIC microcontroller was not based on any comparative research. There are most likely many other microcontrollers available that have the necessary input/output configurations, memory, and timing precision for the task. The PIC was chosen because the Autonomous Vehicles Group at Draper Labs has considerable experience using the PIC microcontrollers on autonomous vehicles. Draper Labs already had the programming components and technical expertise for the PIC16C73A, so choosing this microcontroller benefited the project by reducing development time and promoting uniformity in microcontroller selection.[2]

[2]Two PIC16C73A chips are already in use on the DSAAV as part of the sensor sampling and signal processing board designed by Christian Trott and described in detail in [23].

A brief description of the PIC functions is given here; The assembly language code for the PIC microcontroller is in Appendix B along with a diagram of the PIC pin connections.

2.5.2 Startup

Upon startup, the PIC loads the configuration word into the precision timing module. This configures the timing module for a 500 ps time step with triggering on the laser fire and receive signals. The configuration and variable registers for the PIC are initialized as well.

2.5.3 Sampling

The PIC proceeds to enter the main sampling loop, which is repeated at a rate of 500 Hz. The speed of this loop is regulated using the timing interrupt flag generated by a built-in PIC timing counter. At the start of the loop, the PIC begins sampling the analog input corresponding to the servo position. The analog-to-digital conversion takes multiple clock cycles, so this needs to be started early. Immediately after starting the conversion, the PIC pulses the laser and simultaneously initiates counting on the timing module. Soon after the laser pulse reflection is received by the detector (triggering the stop of the timing module), the PIC receives a signal from the timing module indicating the timing data is available for transmission. The PIC then reads the parallel data outputs of the timing module into its own register memory for the range measurement. The analog-to-digital conversion of the servo position is now finished, and this angle measurement is stored in the PIC's register memory as well. Finally, the PIC sends these digital values to the on-board computer and completes the main loop.

2.5.4 Communication

The PIC communicates with the on-board 486 computer using a RS-232 serial line operating at 38.4 kbs. This serial line is used only for one-way communication, so handshaking is unnecessary, and communication can be done using only the DATA and GROUND lines of the RS-232 interface. To simplify debugging, the laser angle and range data is sent along the serial line as a series of ASCII characters in hexadecimal format. Using ASCII characters simplifies testing of the laser by making it possible to display the laser rangefinder output in any standard ASCII terminal window. Data can be captured into a text file and saved for later analysis in MATLAB

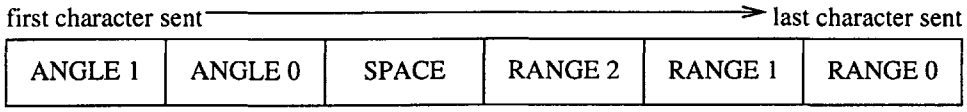

Figure 2-1: PIC output data word

or other signal processing programs. The typical output data word from the PIC is shown in Figure 2-1.

In the case that any portion of laser sampling process causes an error, the PIC indicates this condition by sending the ASCII character "!" to the on-board computer.

One major disadvantage of using ASCII format for sending the scanning laser rangefinder data is that ASCII format is much less space-efficient than binary. The binary data stored in the PIC registers takes up a total of 3 bytes per sample (1 byte of angle data, 2 bytes of range data). Using ASCII characters to represent this data in a readable form takes 7 bytes, as shown in Figure 2-1 above. Thus the PIC is utilizing the serial communications line more than three times as much by as it would be if the data were sent in binary format.

One other handicap of the present communication protocol is that it has no error checking. When a byte is transmitted by the PIC, no checking is done to ensure that this byte has been transmitted correctly. Similarly, there is no error checking to ensure that the bytes received by the on-board computer are identical to the ones that were transmitted by the PIC. Corruption of a single bit in one of the range data bytes can cause a range of 10 ft to be changed to 2058 ft.

Clearly the data should not be sent in ASCII format. In the actual implementation of the helicopter, the PIC code should be modified to send data in binary format with a checksum. This will provide faster, more reliable communications. However the ASCII transmission is much more convenient for debugging purposes, and will continue to be used until the laser rangefinder design has been solidified.

2.6 Performance

One problem with using a laser scanner for object avoidance is the extremely low fanout of a laser beam. While this low fanout allows lasers to have great range, it poses a problem for detecting small objects. Since a laser only gets the range along a single discrete angle in 3D space, there will be inevitably be gaps in the area scanned by the laser. While this is generally not a problem if all objects are sufficiently large relative to the angular resolution of the scanner, smaller objects may be missed

entirely by the laser scanner. In addition, the lack of vertical scanning of the laser makes it particularly poor at picking out objects slightly above or below the scan line.

2.7 Conclusion

Designing a time-of-flight scanning laser rangefinder was an ambitious goal, so it is not surprising that there still are some bugs in this prototype system. This sensor represents cutting-edge technology that will certainly be improved with future work. Both the range and noise rejection of the rangefinder must be improved before it will be suitable for use as an obstacle avoidance sensor on the autonomous helicopter. However, this sensor does provide a starting point from which to design the rest of the autonomous helicopter collision avoidance and mapping system.

Chapter 3

Mapping System Requirements

The scanning laser rangefinder provides the autonomous helicopter's on-board computer with a stream of angle and range measurement pairs. This data needs to be put to use somehow for obstacle detection and localization, as well as the other project goals stated in Section 1.3.1. This section outlines some of the objectives for a mapping system on the Draper Small Autonomous Aerial Vehicle.

This thesis uses the word "map" in a very broad sense. In this thesis, a map refers to *any* representation of obstacles and their locations. A map can be either two-dimensional or three-dimensional, and can represent obstacles in any manner that can be implemented on a computer.

3.1 The Need to Store Obstacle Locations

In representing the range data produced by the laser scanner, it is desirable to store the locations of any objects encountered, as well as note any unoccupied areas that are encountered. Occupied regions should be marked because these are regions where the helicopter can not fly because it would collide with a solid object. Unoccupied regions should be marked because these are regions where the helicopter can fly without danger of colliding with solid objects. In addition, storing the locations of objects will allow the helicopter to use the locations of these objects for other purposes, such as path planning or landmark based navigation.

Path planning is the process of determining a set of points that connect an origin and goal, and is typically performed for the navigation of autonomous vehicles though environments with obstacles. In the absence of any obstacles, a path might take the shape of a straight line connecting the origin and the goal. In the presence of obstacles, path planning becomes more complicated. The autonomous vehicle may want to move along the shortest traversable path between the origin and goal, or might want to move

along a safer route that does not go too close to the obstacles. This thesis does not attempt to delve into path planning for the autonomous helicopter. However future work on the autonomous helicopter will likely involve some sort of path planning, and the stored representation of range information should support this process.

Landmark-based navigation also requires knowledge of obstacle locations. In landmark-based navigation, an autonomous vehicle makes measurements of range and angle from itself to a landmark object. The autonomous vehicle is assumed to have previous knowledge of the exact location of this landmark object, possibly through a database or operator assistance. Using the location of the landmark object and the range and angle measurements to this object, the autonomous vehicle can calculate its own position and orientation with some ambiguity. The ambiguity can be resolved by making additional angle and range measurements to other landmark objects, and then reconciling this data with the original calculations. In two dimensions, this process is commonly known as triangulation. This thesis does not attempt to implement landmark-based navigation. However it is important that it be possible to perform landmark-based recognition as an addition to the mapping system.

3.2 The Need for Filtering

As mentioned in Section 2.6, the data produced by the scanning laser rangefinder has inherent errors associated with it. Errors in the angle and range measurements can occur because of reflections of the laser, discretization errors, transmission errors, and malfunctions of detector, timing module, or PIC, among other things. The range data needs to be filtered in some way to provide a useful map. An ideal filtering system would be able to gracefully handle both the drastic errors caused by laser reflections and the smaller quantization errors and noise in the measurements. Error filtering should be done with consideration for the ultimate scanning laser rangefinder goals of obstacle avoidance.

Error filtering should also be performed according the the specific types of errors observed on the scanning laser rangefinder data. However this information was not available at the time when the mapping system was being designed. The errors present in the scanning laser rangefinder measurements are described and categorized in [18].

3.3 The Need for Confidence Representation

Closely linked with the need for data filtering is the necessity of some sort of confidence measurement with respect to the location of obstacles. Consider a situation where the laser rangefinder reports 15 separate times that the range to the nearest object at angle 0° from the helicopter is 35 ft. The autonomous helicopter should have more confidence in this measurement than if the laser rangefinder were to report two measurements of 35 ft and one measurement of 20 ft. It is important for the mapping system to store confidence values because this will determine which regions the helicopter flies into and which regions it avoids. An autonomous helicopter programmed to act in a optimistic and risky manner will fly into regions where confidence is low. A more cautious approach might require the helicopter to have high confidence in any region before entering it. The mapping system should provide some way of representing different degrees of confidence in obstacle locations.

One possible mission that has been listed for the DSAAV is target location verification. In a mission of this type, the autonomous helicopter will be given an a priori map of a region and asked to verify the positions of key targets within this region. If the positions of these targets have moved, then the helicopter will need to gradually decrease the confidence in each target's old position and increase the confidence in each target's new position. This is only possible with an obstacle representation that supports varying levels of confidence.

3.4 Uncertainty in Helicopter Position and Orientation

In addition to the errors in the range and angle measurements performed by the scanning laser rangefinder, there is also a degree of uncertainty in the position, orientation, and velocity of the autonomous helicopter. Put simply, the autonomous helicopter does not know exactly where it is located and how it is oriented. The state of the helicopter is determined entirely by sensor measurements. Since there will be errors these measurements, there is also error in the autonomous helicopter's estimate of its own state.

The helicopter's state is used in conjunction with the laser range and angle measurements to compute the positions of obstacles.[1] Since there is uncertainty in both the laser rangefinder measurements and the helicopter state, this means there will be compounded uncertainty in each obstacle location. The chosen mapping system

[1]The calculation of obstacle locations is described in Chapter 6.

should provide a mechanism or structure for dealing with this uncertainty in obstacle locations.

3.5 Three-Dimensions

The obstacle representation needs to represent obstacles in three dimensions because the autonomous helicopter can move in all three dimensions. A two-dimensional obstacle representation that only stores objects that are at the same height as the helicopter would prohibit the autonomous helicopter from ever moving safely in the vertical direction. Imposing such a limitation is unacceptable because it limits the potential functionality of the autonomous helicopter. Three-dimensionality is explicitly noted as a requirement because many autonomous rovers have successfully used two-dimensional obstacle representations for similar autonomous goals. Rovers can afford to use two-dimensional representations because their movement is restricted in the vertical dimension (rovers can't fly like a helicopter can).

3.6 Varied Environment

The autonomous helicopter will be flying in an incredibly varied environment. The selected mapping system should be able to represent the large faces of office buildings, the rough outlines of trees, and all varieties of ground terrain from snow-covered mountains to desert sand. It should be able to represent equally well both the open expanses of the desert and the multitude of objects present in an urban environment. The obstacle representation system should be either universal, or easily adaptable to a wide variety of obstacles.

3.7 Large Volume

Missions for the autonomous helicopter will likely span a large area. A typical mission might involve flying a distance of 2 miles to a nearby town, verifying the locations of certain military targets in this town, and then flying 2 miles back to the launch point. In order to record obstacle locations over this entire range, the autonomous helicopter will need to employ a compact representation of obstacles and their positions.

3.8 Medium Precision

At some point in the future, the DSAAV will need to demonstrate precision flying in an obstacle-rich environment. While this thesis does not address the topic of obstacle

avoidance algorithms, the mapping system should be useful for this. The autonomous helicopter should be able to notice and avoid nearby obstacles by maintaining precise measurements of obstacle locations. While it is difficult to set exact numbers on how much precision is needed to fly the helicopter, the obstacle representation should have a resolution of better than 3 feet.

Landmark-based navigation, one of the other desired uses of the obstacle representation, also requires precise measurements of obstacle locations. Therefore it is desirable to provide as much precision as possible in the obstacle representation. The scanning laser rangefinder itself sets a practical limit on resolution selection at 3 inches, since there is little point in designing an obstacle representation with better resolution than the rangefinder itself.

3.9 Fast Obstacle Avoidance

Land rovers and other slow moving vehicles can afford to spend minutes calculating obstacle locations before they proceed. The Draper autonomous helicopter has no such luxury. The helicopter needs to be able to update its obstacle representation and avoid oncoming objects while flying as fast as 35 mph.

If obstacle avoidance is to be done through the mapping system, then this means that updates to the map must be very quick. Data received from the scanning laser rangefinder should be incorporated into the obstacle representation immediately, so that obstacle avoidance algorithms can begin to act as soon as possible. In addition, the method for adding range data to the obstacle representation should be streamlined as much as possible to ensure fast execution.

3.10 Limited Processing Power

In the current version of the helicopter, only 25% of the processing time on a 486 100 MHz computer is available for use by the laser scanner software. This will almost certainly *not* be enough computation power to implement a suitable mapping system.

Rather than allowing this lack of processing power to limit algorithm development for the mapping system, we decided to proceed under the assumption that the autonomous helicopter's on-board computer will be upgraded in the future. This means that algorithms should be designed to run as fast as possible, but the current on-board processing power should not overly affect development choices.

3.11 Limited Memory

The current on-board computer has approximately 16 MB of RAM available for storage of the mapping structure, though this can be expanded to 64 MB if necessary. At the initiation of the scanning laser rangefinder project there was no hard disk, but a 20 MB hard drive has since been added to the on-board computer system.[2] Most of the RAM is available for use by the mapping system software. A good design for the mapping system system should work with the current memory available on the computer, while still allowing for improvements if the on-board memory increases.

3.12 Versatility

While the mapping system is initially going to be used with just the scanning laser rangefinder, it should be versatile enough to be used with other sensors as well. Draper anticipates that sometime in the future, better ranging sensors will become available to the helicopter. These sensors may scan in more directions, use different technology, and may either replace or be used in conjunction with the laser rangefinder. Possibilities for alternative sensor technologies include sonar, phase-detection laser, and ultra-wide-band radar systems. [3]

It should also be possible for two autonomous helicopters to independently add range information to the same map structure, thereby creating a map that is useful to both of the helicopters. A well-designed mapping system should be universal enough that it can be adapted to different sensors or multiple sensors without a complete redesign of the system. It is not acceptable to redesign the mapping system for new sensors, because this will take extra time and will render useless any work based on the original mapping system. The mapping system must provide a solid foundation that will be useful in a wide range of applications.

3.13 Accessibility

In order to be useful for tasks such as fast obstacle avoidance, the mapping system needs to be easily accessible by other on-board software. It is anticipated that the autonomous helicopter will make use of the mapping system by running independent programs that make use of the obstacle data. One process might use the map for

[2]The on-board computer currently runs the QNX operating system. Software is loaded from the ground station to the on-board computer and execute via remote commands from the ground station.

[3]Long Phan's thesis, reference [18] discusses alternative sensor technologies in detail.

path planning, another might use it for object recognition. All of the processes will need the obstacle representation to be easily readable.

Also, the DSAAV ground station currently displays information about the state of the helicopter. This information is sent from the autonomous helicopter to the ground station using a radio modem and includes the current helicopter position and orientation, the on-board battery voltage, the current guidance mode, and a few other things. When obstacle interaction features are added to the helicopter, it will be desirable to display some portion of the obstacle representation on the ground system. (Because of bandwidth limitations in the radio modem, it is unlikely that it will be possible to transfer the entire map to the ground station.) While it is not necessary to implement transmission of the map at this point in time, it should be designed so as to be easily communicable to the ground station.

3.14 Summary of Objectives

This is an extensive and ambitious list of requirements. Spanning large volumes of space with good precision generally requires large amounts of memory, something that is not available on the helicopter. In addition, some of the requirements are disjoint. For example, fast obstacle avoidance would be achieved by reacting to an obstacle as soon as a hit is noted, but noise rejection would be improved by waiting for additional measurements to verify the accuracy of any one sample. Different types of map structures work better in different cases, and the next chapter gives an overview of the design choices made in selecting the mapping system.

Chapter 4

Previous Work

Very little work has been done in producing three-dimensional (3D) maps for use by autonomous aerial vehicles. Therefore it is necessary to look elsewhere for insights on how to design a 3D mapping system. This chapter gives a brief overview of some alternative mapping strategies that might be useful for building an obstacle representation on the autonomous helicopter.

Quite a bit of research has been done in developing two-dimensional (2D) maps for use by autonomous land vehicles. Section 4.1 outlines two basic types of maps that have been implemented successfully in two dimensions: object lists and certainty grids.

Autonomous underwater vehicles operate in a three-dimensional environment that is similar in some ways to that which will be encountered by the DSAAV. Section 4.2 gives an overview of the height field mapping structure that is commonly used in underwater vehicles.

The discipline of computational geometry [3] has developed methods to efficiently represent complex three-dimensional objects for display on computer screens. Section 4.3 describes a few map-like representations that can be borrowed from the field of computer graphics.

4.1 Two-dimensional Maps for Land Vehicles

While the three-dimensional mapping problem is different from the two-dimensional problem, many two-dimensional strategies can be extended to work in three dimensions as well. This section gives an overview of mapping representations by outlining some 2D mapping strategies that have been used successfully in autonomous land robots. This overview of 2D mapping strategies provides a good background for work in developing a 3D mapping system.

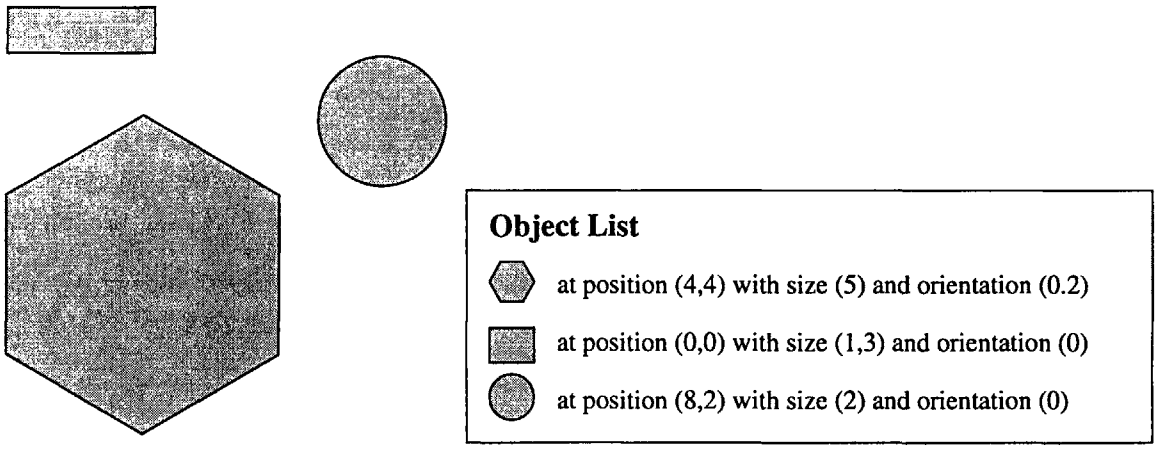

Figure 4-1: A Sample Object List

Two-dimensional maps generally fall into one of two categories: object lists and certainty grids. Object lists store a list of identified objects and their positions. Certainty grids divide up an area into a grid of cells and then note the occupancy of each of these cells. Both of these representations are theoretically extensible to three dimensions.

4.1.1 Object Lists

Object lists store a list of all objects in a specific area that a robot has encountered. When the robot encounters a new object, it classifies this object, notes its location, and adds this object to the object list. This is just how people remember areas. For example, an object list representation of a soccer field would be just a list of two objects, the goals at either end of the field. There's nothing complicated about remembering the wide open playing field, so people don't bother thinking about it, and neither would a robot that maps an area using an object list.

Figure 4-1 shows an object list representation for a few simple geometric shapes. Notice how each object is categorized according to its shape, and then stored in the object list along with other characteristics such as its position, size, and orientation.

Object lists are good obstacle representations for robots because they don't take up very much memory. In the case of the huge soccer field, the robot only had to remember the two goals. Of course, in order for the object list to be any use, the robot needs to know what a soccer goal is. This is one big disadvantage of object lists: *all the objects must be recognizable.* Otherwise the robot won't be able to classify the new objects and store them in the list.

One way around this problem is to classify two-dimensional objects as groups of triangles or rectangles, which in turn leads to another disadvantage of object lists for robots: *classification of objects can be difficult*, especially in environments with irregularly-shaped obstacles.

Another disadvantage of obstacle lists is that it is ambiguous how one would use an obstacle list to represent unknown and open areas. Most of the autonomous robots that use obstacle lists do not maintain any sort of uncertainty representation in their lists. Instead, they simply store the best estimate of obstacle locations and use this for all additional calculations. Obstacle lists are seldom (if ever) used for filtering range measurements when there is no prior knowledge of obstacle types and locations.

However, for all their disadvantages, object lists are very easy to use once they've been created, and have been used extensively in the development of obstacle avoidance and path planning algorithms. [14] [7] Since obstacles and their locations have already been identified and stored in the list, calculating interactions with these obstacles is simplified. In most of these cases, an obstacle list is loaded during the initialization of the autonomous land vehicle and then used for guidance purposes as the vehicle moves through the known environment. Range measurements to obstacles are used to update the position of vehicle in the map, and not vice versa. Thus while an object list is not a good representation for filtering range information, it is very useful to path planning, obstacle avoidance, and landmark-based navigation algorithms.

4.1.2 Certainty Grids

Certainty grids are the second major type of map that is commonly used on autonomous land vehicles. In a certainty grid representation, all of free space is divided into a grid. Each cell in this grid contains a number indicating the probability that this cell is occupied. Cells that are definitely open are assigned a probability of 0%, and cells that are definitely occupied are assigned a probability of 100%. Typically cells will start out with a probability of 50% and then are updated accordingly for sensor hits and misses. For example, if the laser senses an object in a cell that is originally unknown (50%), the occupied probability of that cell might be updated to 75%. Once a few sensor readings have been taken, the robot can use the grid probabilities to decide which cells are occupied and which are unoccupied. Occupied cells are treated as obstacles and can avoided in path planning or grouped with neighboring cells for object recognition purposes.

Figure 4-2 shows a simple certainty grid. In this figure, occupied cells are colored black, unoccupied cells are colored white, and shades of gray are used to represent

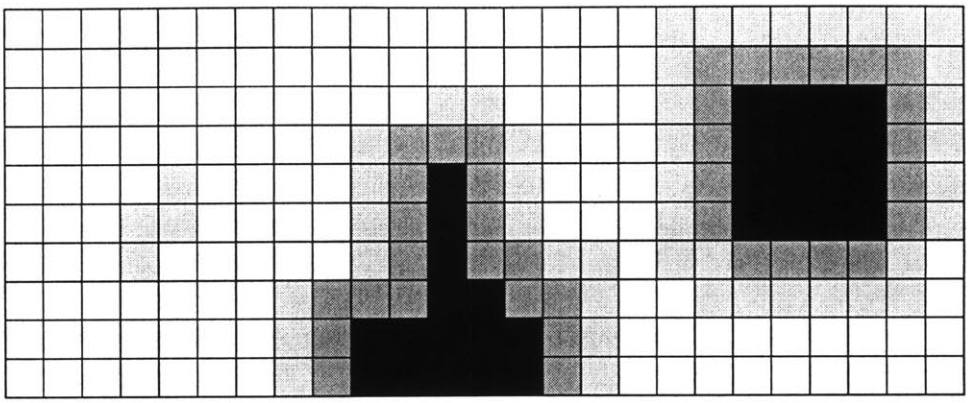

Figure 4-2: A Basic Certainty Grid

cells with more ambiguous certainty values. Notice how there is a region of uncertain cells surrounding each of the black objects that have been marked in the certainty grid. These uncertain cells are the result of errors in the sensor reading, and crudely represent the associated uncertainty in the shape and position of the obstacles. Erroneous sensor from of reflections and noise can also cause there to be "ghost" objects in the certainty grid. The four light gray cells on the left side of the map represent a ghost that has been mostly filtered by the certainty grid.

One big advantage of certainty grids is their robustness. A few bad sensor readings can be filtered out reliably, and therefore won't throw off the obstacle detection capabilities of the grid. Also, unlike obstacle lists, there is no need to identify what an object is before noting that it needs to be avoided. This mean that certainty grids can function well even in the presence of large amounts of noise and many sensor errors. [15] [5] [16]

Certainty grids are also extremely versatile. While obstacle lists need to identify objects to represent them, certainty grids can represent identifiable and unidentifiable objects equally well. There is no need for a certainty grid to be loaded with a priori information (though this is certainly possible), nor is there any need to make assumptions about the size, nature, or orientation of the obstacles that the autonomous vehicle will encounter.

One disadvantage of certainty grids is that they must discretize range information in order to store it in the grid. The discretization and data addition process can smear features and thereby make feature detection more difficult. By filtering data in a certainty grid, the measurement uncertainty (the error in the laser rangefinder measurements) and navigational uncertainty (the error in the autonomous vehicle

Certainty Grid	Object List
Makes a grid of free space and assigns probabilities that each cell is occupied.	*Makes a list of obstacles and their positions.*
- No concept of any obstacle	+ Can identify obstacles
+ Less computation in building map	- Need to identify edges of obstacles
+ Can note unexplored and open areas easily	- More complicated to note unexplored and open areas
- Navigation feedback, object recognition not well supported	- Navigation feedback, object recognition are easy once the list is made
+ More robust, can easily filter bad readings	- Less robust, creates objects from bad data
+ Easily extensible for different sensor types	- Difficult to extend to different sensors
+ Versatile for varied environments	- Modifications necessary to add new objects
- Memory intensive	+ Less memory intensive
- Needs bounded space	+ Easily expanded in space

Figure 4-3: Comparison of Object List and Certainty Grid

position) are combined. This means that it is no longer possible to perform landmark-based navigation measurements with the same degree of precision.[20]

Robustness and versatility are both important qualities in the obstacle representation implementation on the DSAAV, where data will have lots of noise and many obstacles will be unrecognizable. In addition, certainty grids are easily modified for use with different sensors or even multiple sensors.[15] Since there is a distinct possibility that the laser will be replaced or redesigned, it is important to have a software mapping structure that can accommodate these changes. A summary comparison of obstacle lists and certainty grids is shown in Figure 4-3.

The choice between using a certainty grid and an obstacle list depends on the application in question. For widely spaced objects that are easily identifiable, an obstacle list is the obvious choice. For bounded environments where there are complicated obstacles or lots of noise, certainty grids are clearly superior. For all other environments, the decision is not nearly so simple. Researchers have developed further storage variations that combine various features of both methods, which add further mapping options. The strengths and weaknesses of each representation must be carefully weighed against the assumptions that can be made about the robot and environment.

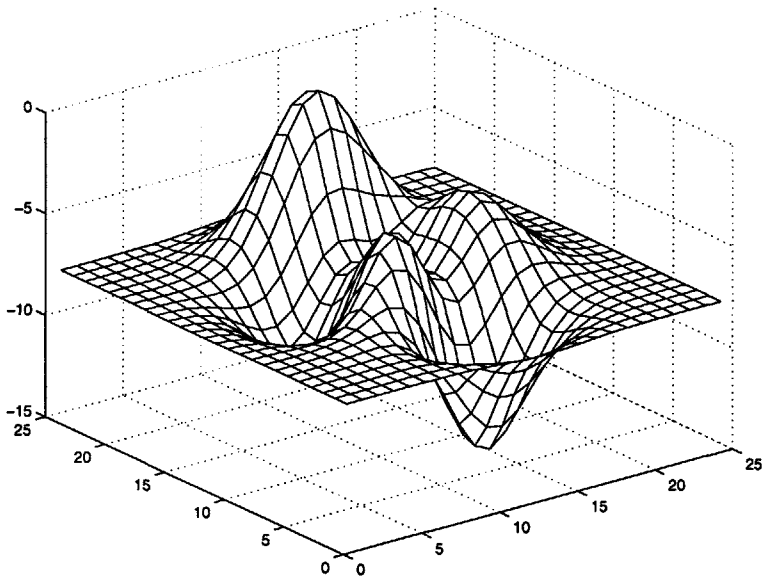

Figure 4-4: A Sample Height Field

4.2 Height Fields for Underwater Environments

Autonomous underwater vehicles have existed for some time, and operate in an environment similar to that of the autonomous helicopter in that it allows freedom of motion in three dimensions. By making sonar measurements of the depth of the ocean at certain locations, water vehicles build 3D maps of the ocean floor. This process is known as bathymetry and the resulting 3D maps are called bathymetry maps. Bathymetry maps typically are stored as height fields.

A height field is created by dividing up the entire ocean surface into a two-dimensional grid of cells (the ocean surface is assumed to be a flat plane). Each point in the grid is indexed with an (x, y) coordinate and stores the depth of the ocean at the specified location. This means the height field measurement stored at grid location (x, y) is depth of the ocean beneath the surface position (x, y). Thus the grid represents a discrete 2D sampling of ocean depths.

Figure 4-4 shows a simulated height field. Notice how the drawn grid dips down for greater depths in the height field and peaks when the stored height field depth is at a minimum.

Some bathymetry maps that are produced by underwater vehicles include representations of the uncertainty in the depth. For each grid point in the height field, an additional value is stored indicating the uncertainty in this measurement. This uncertainty is typically stored as a standard deviation or variance in the corresponding

depth measurement.[24] By storing an uncertainty value with each depth measurement, it is possible for autonomous underwater vehicles that use height fields to filter out noise and other errors from their depth measurements. As more measurements are taken of the ocean depth at a specific grid points, the standard deviation and variance of these measurements will decrease, indicating that there is greater certainty that the stored depth is correct.

One notable limitation of height fields is that they make the assumption that $z = f(x, y)$ is single-valued. This means that it would be impossible to use a height field to represent any sort of cave or other interior region where there is unoccupied space beneath occupied space. Since the height field only stores the minimum depth at each grid position, all objects beneath this are assumed to be occupied, which may not true in a complicated environment that includes caves or overhangs.

4.3 Hierarchical Representations from Computer Graphics

The fields of computer graphics and autonomous vehicles overlap in their need to store information about object locations within large areas and volumes. In the development of autonomous vehicles, one goal is to represent the obstacles encountered by a vehicle moving through a region. In the study of computer graphics, the goal is to represent objects so that they can be drawn quickly and efficiently on a computer screen. Both areas of study have successfully used hierarchical representations to improve the performance of their object representations.

One basic type of hierarchical object representation is a quadtree. Quadtrees have been used in both computer graphics and autonomous vehicles to efficiently demarcate two-dimensional areas.[21][10] Section 4.3.1 describes the application of a quadtree to improve a certainty grid.

There are a number of quadtree-like structures that have been used successfully in computer graphics but have seldom been applied to mapping systems for autonomous vehicles. Sections 4.3.2 and 4.3.3 briefly describe two of these other structures.

4.3.1 Quadtrees

The basic object list and certainty grid representations used in autonomous vehicle mapping systems are often augmented through the use of a tree hierarchy. In an object list, a tree hierarchy is often used to sort the objects according to their locations. This speeds up the process of searching through the object list to find a desired object. In a certainty grid, a tree hierarchy is often used to reduce storage space in memory by

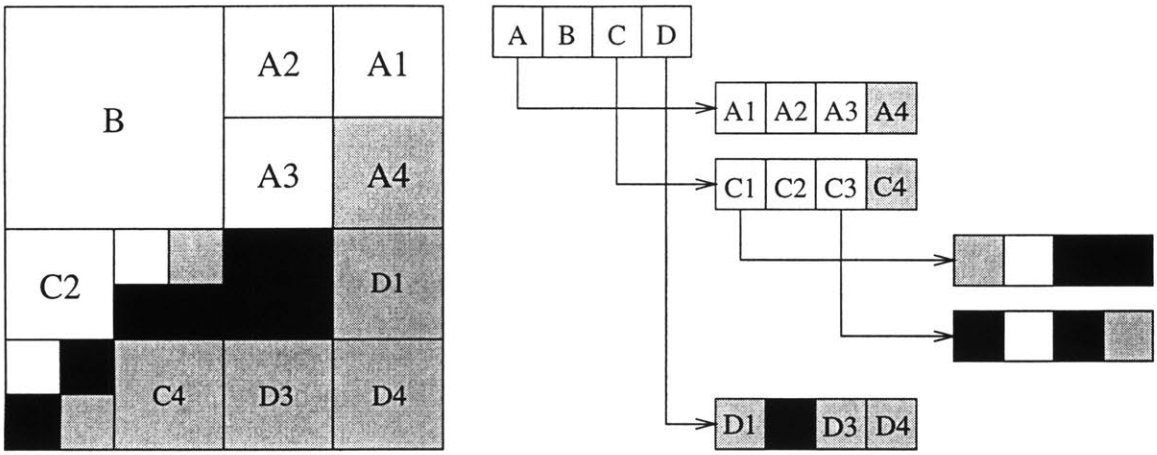

Figure 4-5: A Quadtree Subdivision

giving the grid a variable resolution. The hierarchy allows areas that are uniformly occupied or uniformly unoccupied to be represented with low resolution, whereas more complicated areas can be represented with better resolution. In the following discussion, a quadtree for a certainty grid is described, but the principle is also equally applicable to object lists.

To build a quadtree, we first select a bounded square region to be mapped. This region is then divided up into four cells, just like a 2x2 certainty grid. Each of these cells can then be divided up into four smaller cells, forming the second level of the hierarchy. These smaller cells can in turn be recursively divided to form even more precise levels in the hierarchy. However, often it is not necessary to divide all parts of the map with this level of precision. While there are generally some portions of the map area that require high precision, there are also generally some portions of the map where large, imprecise cells work perfectly well. For these uniform areas of the map where large cells suffice, quadtrees halt their subdivision and store only the large cells. This saves on memory and makes quadtrees very appealing for large map areas.

Figure 4-5 shows a quadtree representation for a small area. On the left is the area breakdown, and on the right is the quadtree itself. This quadtree breaks up areas according to their color: black, white, or gray. Notice how area B remains undivided, whereas area A is forced to be subdivided by cell A4 being gray instead of white. Similarly, area C is subdivided twice because of the variety of cell colors in this area. Also note the decreased memory storage- the quadtree used only 24 cells, approximately one third of the number of cells that would have been used by a

certainty grid representation of the same area.

Quadtrees are generally built using recursive update methods. That is, the tree will start out very simple, with only one level of hierarchy, and will become more subdivided as more and more data is added to the tree. Since each level is designed the same as the previous one, the same function can be used to subdivided cells at all levels of the tree. Most quadtrees also set a maximum tree depth or minimum cell size to prevent the tree from getting overly large and complicated.

There are numerous variations on quadtrees that add nuances to the subdivision process.

4.3.2 Octrees

The three-dimensional sibling of the quadtree is known as the octree, because it breaks volumes into 8 cubes separated by the x,y,and z planes. Octrees are therefore useful for partitioning three-dimensional objects, but are seldom used because of their high branching factor (8). Since each subdivision of an octree creates 8 new cells, octrees can take up a lot of space in memory even when they are just a few layers deep.

4.3.3 Kd-trees

Kd-trees improve upon quadtrees and octrees by reducing the branching factor of the hierarchical representation. In a kd-tree, cells are subdivided into two parts rather than into four parts like a quadtree, or eight parts like an octree. Minimizing the branching factor of a hierarchical representation helps to minimize the number of total number of cells in a tree by subdividing the tree only when absolutely necessary.[10][3]

Kd-trees can be applied to both two and three dimensions. In two dimensions, kd-trees subdivide cells by breaking them in half either vertically (along the y axis) or horizontally (along the x axis). Figure 4-6 shows the subdivision of a 2D kd-tree as necessary to represent the colored gray square A. Kd-tree subdivisions are always performed parallel to the coordinate axes, so all cells in a 2D kd-tree will be rectangles. This makes it easy to store and draw kd-trees.

There is no single rule for deciding how a kd-tree should be split. Some kd-trees will split cells along their longest dimension. Other kd-trees will split cells along alternating dimensions. Still others may split cells so as to minimize the depth of the tree. The correct split rule for a kd-tree depends on the application.

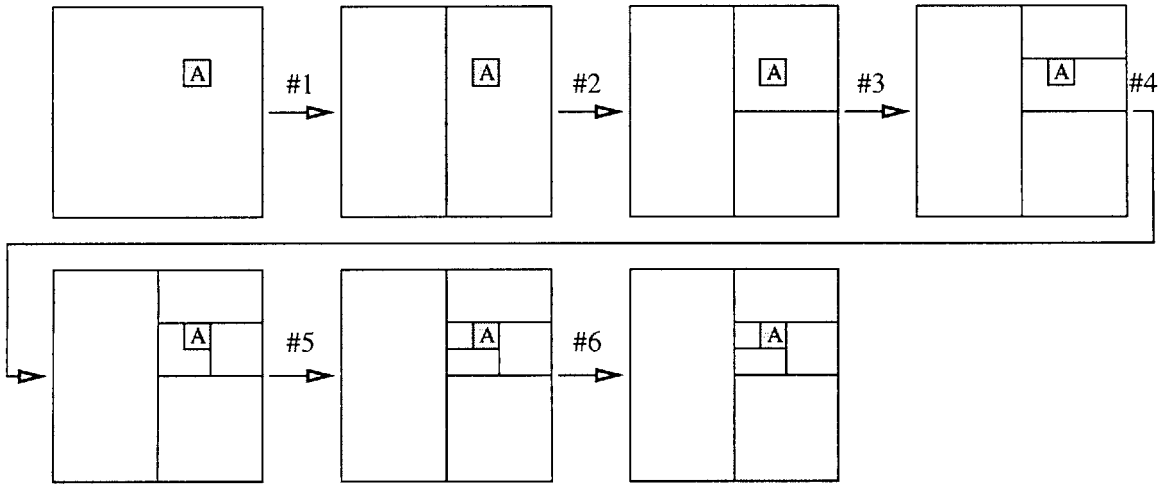

Figure 4-6: Kd-tree Subdivision

Chapter 5

Selecting Mapping System

Designing a specialized obstacle representation such as this one is a highly iterative process. I started out with a simple 3D certainty grid, and then built upon this to create a complete mapping system

5.1 Starting With a Certainty Grid

A certainty grid representation was chosen over an object list primarily because of its robustness and versatility, both of which are very important on the helicopter laser scanner platform. It can easily be used in any sort of environment, and is applicable with minimal modifications to all types of sensors. In addition, it is simple and straightforward to note occupied, unoccupied, and unexplored areas in a certainty grid. Cells can be marked quickly and easily according to the data presented by the autonomous helicopter, and filtering of bad range and angle values is done automatically. This makes a certainty grid a perfect foundation on which to build a mapping system.

Object lists are powerful representations, but lack the robustness necessary to be the foundation of the autonomous helicopter's mapping system. Object lists can not easily be used in environments with varied and unidentifiable obstacles, since each obstacle must be identified before it is added to the list. Identifying obstacles is also slower, and it is unclear how one would use an object list to perform real-time filtering of bad data values. Filtering bad data samples and representing volumes that are unknown or ambiguously occupied is more difficult with an object list than with a certainty grid. Since one of the most important objectives of the obstacle representation is filtering the error-prone range data, an object list was deemed to be less appropriate for use on the autonomous helicopter.

Note that the selection of a certainty grid as the basis representation for the map-

ping systems does not necessarily eliminate object list representations from all parts of the autonomous helicopter software. Other programs that perform tasks such as object recognition and path planning could use the certainty grid as a basis for creating their own object lists. This way, the object list is generated from the certainty grid-filtered data, which makes for better object recognition. This extensibility is yet another reason why the certainty grid is an excellent fundamental structure for the DSAAV mapping system.

5.2 Adding the Kd-tree

The greatest disadvantage of the certainty grid representation is that it takes up quite a bit of space in memory. A cube of cells with 1000 cells on a side, with each cell storing a 1 byte occupancy value takes up 800MB of RAM![1]

Since much of the volume that the helicopter will be sensing is open air, this is an incredible waste of memory. In addition, the bounded nature of the certainty grid makes it less than ideal for the long range missions planned for the helicopter.

One logical improvement to the certainty grid approach is modify the certainty grid into a octree or a kd-tree. This hierarchical subdivision will reduce the memory requirements by grouping together the large, open-air volumes into single cells, and will reduce the memory demands for unexplored areas as well. Using a hierarchical subdivision also helps to solve the certainty grid's bounded-volume problem, since the initial volume can be set to be huge with only a minimal penalty in memory requirements. As the scanning laser rangefinder provides data about a small portion of this larger bounding volume, the volume will be subdivided accordingly in the target region.

Using the kd-tree representation for the volume directly in front of the helicopter is inefficient. New data points are constantly being added to the entire volume directly in front of the helicopter, so this portion of the kd-tree is always at its most precise cell size. In this case, the kd-tree is no better than a certainty grid, and is actually a little worse because the linked list nature of the tree will increase cell access time over that of a certainty grid. The laser rangefinder produces range measurements at a rate of 500 Hz, and each range measurement will affect the stored values of multiple cells, so there will be thousands of cell reads and writes every second. Thus a small

[1] I use the term cell to refer the an element of 3D space in the shape of a rectangular prism or box. This is the terminology commonly used in robotics papers discussing certainty grids (cite examples). Some computer graphics texts use the term "voxel" (short for "volume element," like a 3D pixel) to refer to the same concept. In this thesis, a "cell" is equivalent to a "voxel."

increase in access time for a single cell will make significant difference in the overall performance of the program.

It's much more straightforward to add the new data to a simple 3D certainty grid. A better solution would be to somehow use a simple certainty grid for adding and filtering new range measurements, and a kd-tree for the rest of the obstacle representation.

5.3 The Combination Mapping System

This thesis presents a mapping system that consists of two separate obstacle representations. The first of these obstacle representations is the "local map." The local map is a simple certainty grid that is maintained in the helicopter frame of reference. This means that the local map is always local to the helicopter; as the autonomous helicopter moves, the origin and frame of reference of the local map move also. Measurements from the scanning laser rangefinder are added to the local map immediately after their transmission to the on-board computer, so the local map is always current. Since the local map only represents the volume immediately surrounding the helicopter, it can be made with relatively small extent and with good resolution. The local map is described in detail in Chapter 6.

Periodically the contents of the local map will be copied to a kd-tree representation for efficient, non-localized storage. This kd-tree will be in the ground frame of reference and will henceforth be referred to as the "global map." The global map volume will be large enough to encompass the entire range of the helicopter on the current mission, and it is expected (but not required) that the resolution of the global map will be coarser than that of the local map. Like the local map, the global map maintains confidence values indicating the occupied nature of each cell, so that unknown areas in the local map are translated accordingly. The global map is described in detail in Chapter 7.

Figure 5-1 shows a block diagram which puts the mapping system in perspective. The scanning laser rangefinder sends data to the on-board computer, which receives this data and inserts it into the local map. In order to insert the rangefinder data into the local map, the insertion function also needs to know the position and orientation of the helicopter, which are supplied by the previously-existing navigation code. The global map is then periodically updated from the local map. The information stored in the local and global maps can then be used for other purposes such as path planning, obstacle avoidance, landmark-based navigation, and object recognition. Completed

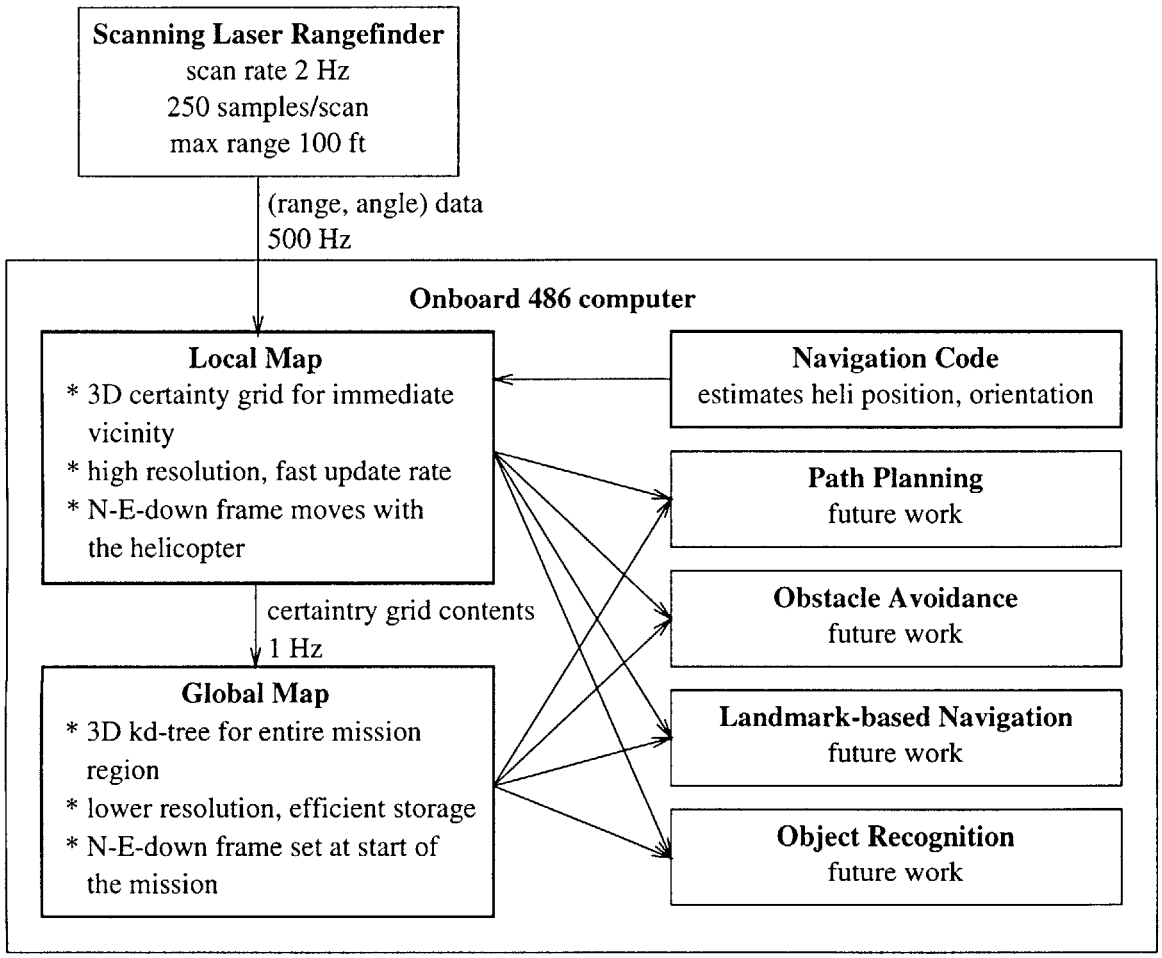

Figure 5-1: Overview of the Mapping System

modules are outlined in bold. The future work necessary to implement these functions is discussed in Chapter 9.

5.3.1 Advantages of the Combination System

There are numerous benefits associated with using these two obstacle representations instead of just one. By using the certainty grid for the local map, we always have a good resolution, quick access obstacle representation for adding and filtering the scanning laser rangefinder data. Since the local map is kept current and relative to the helicopter, it can be used for all obstacle avoidance maneuvers. The kd-tree complements this functionality by providing efficient storage over the large volume that the helicopter will fly through during a mission. The global map can also be used for path planning and combining the mapping efforts of multiple helicopters.

One other added benefit of the local map is that it allows the helicopter to treat measurement noise separately from position uncertainty. Since the laser scanner is mounted on the helicopter, all range measurements to obstacles are made relative to the helicopter's actual position, not where the helicopter thinks it is (unfortunately these aren't always the same). Measurement noise will be included when adding the scanner data to the certainty grid local map, but since this map is relative to the helicopter, there is no need to account for the uncertainty in the position of the helicopter. The uncertainty in the helicopter position will matter over time however, since the errors in measuring the displacement of the local map will result in incorrect shifting of the local map. Similar logic was used by Smith et al as part of a mapping systems for an underwater autonomous vehicle in [20]. See Chapter 9 for a discussion of possible improvements to the mapping system that will represent this error more appropriately.

5.3.2 Disadvantages of the Combination System

Of course, there are also disadvantages to using two maps as opposed on one. Using two maps to represent overlapping volumes means that there is a duplicate representation of the volume covered by the local map. This means that some space is wasted, because a single map would only maintain one representation of the volume. The memory performance of the mapping system is discussed in Chapter 9. A second disadvantage of using two maps is that both maps need to be updated. In the case of the mapping system described in this thesis, this means that data must periodically be transferred from the local map to the global map. This takes up extra processor time. The process of transferring data from the local map to the global map is

described in detail in Chapter 7.

Chapter 6

The Local Map

The first part of the mapping system described in this thesis is the local map. The autonomous helicopter maintains a small, precise local map to represent all obstacles in its immediate vicinity. This map is designed to be used for quick accumulation and filtering of data from the scanning laser rangefinder and to provide a basis for future obstacle-relevant software on the helicopter.

The local map consists of a 3D certainty grid[1] oriented in the north-east-down frame of reference, and is designed to be centered on the autonomous helicopter at all times. As the helicopter moves, the cells of the map are shifted accordingly. When data is received from the scanning laser rangefinder, it is placed into the local map by incrementing the certainty estimations in each of the map cells. Over time, the certainty grid will process all the information from the scanning laser rangefinder to produce a discretized map of obstacle locations in the vicinity of the helicopter. The code for the local mapping structure is included in Appendix D. The C code that processes the local map is included in Appendix C.

6.1 The Map Structure

The certainty grid is implemented as a three-dimensional array of cells. All cells must be cubes, and all the cells must be the same size. The extent of the map is determined by the size of the cells and the number of cells along each dimension. Both of these options can be configured while the software is running, so that the helicopter can adjust its local map according to the current situation. This flexibility of the size and resolution of the mapping structure is an important feature that is discussed further in Chapter 9.

Figure 6-1 shows a sample local map. In this example, the local map has 6 cells

[1] Certainty grids are introduced in Section 4.1.2.

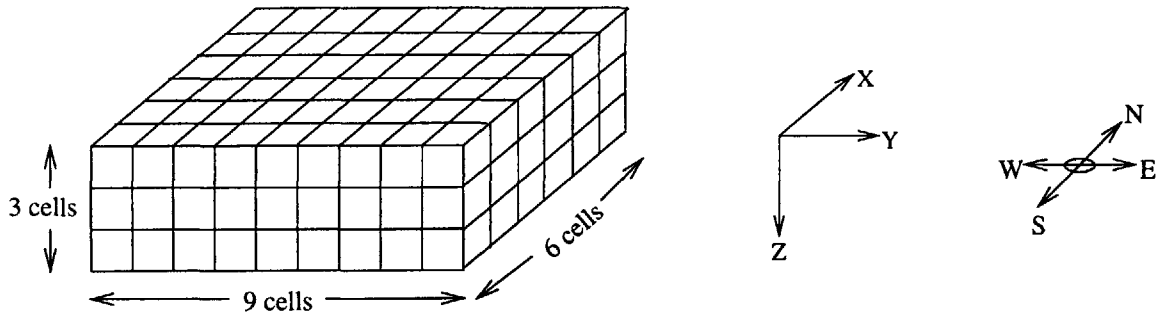

Figure 6-1: A 3D Certainty Grid

along the x dimension, 9 cells across the y dimension and 3 cells across the z dimension. If each cell has an edge length of 3 feet, then the local map will span a length of 18 feet in the x dimension, 27 feet in the y dimension, and 9 feet in the z dimension. Note that the 3D certainty grid shown in Figure 6-1 is only an example. On most missions, the helicopter will be flying at a roughly constant height and scanning the laser rangefinder horizontally. In this case, the x and y dimensions of the local map will be much longer than the z dimension.

6.1.1 Position and Orientation

The 3D certainty grid is oriented in the north-east-down frame of reference. The north-east-down frame of reference refers to the right handed coordinate system consisting of the positive x-axis extending to the north, the positive y-axis extending to the east, and the positive z-axis extending downwards. Unit vectors for this coordinate system are shown on the right side of Figure 6-1. Although the map shifts as the helicopter moves, the orientation of the local map does not change. The axes and cells of the local map are always maintained in the north-east-down coordinate frame.

While the orientation of the local map does not change, the position of map will change as the helicopter moves. Figure 6-2 shows a top view (from $z = -10$) of the movement of helicopter and local map over time. At time t=0, the certainty grid is in the southwestern corner of the map. As the helicopter (marked by the black dot in the center of the certainty grid) flies northeast towards building #1, the certainty shifts accordingly so that the helicopter remains in the center of the local map.

The position of the certainty grid in the ground frame of reference is stored along with the as a (x,y,z) coordinate. This coordinate is calculated from the autonomous helicopter's estimate of its own position and from the offset of the helicopter on the

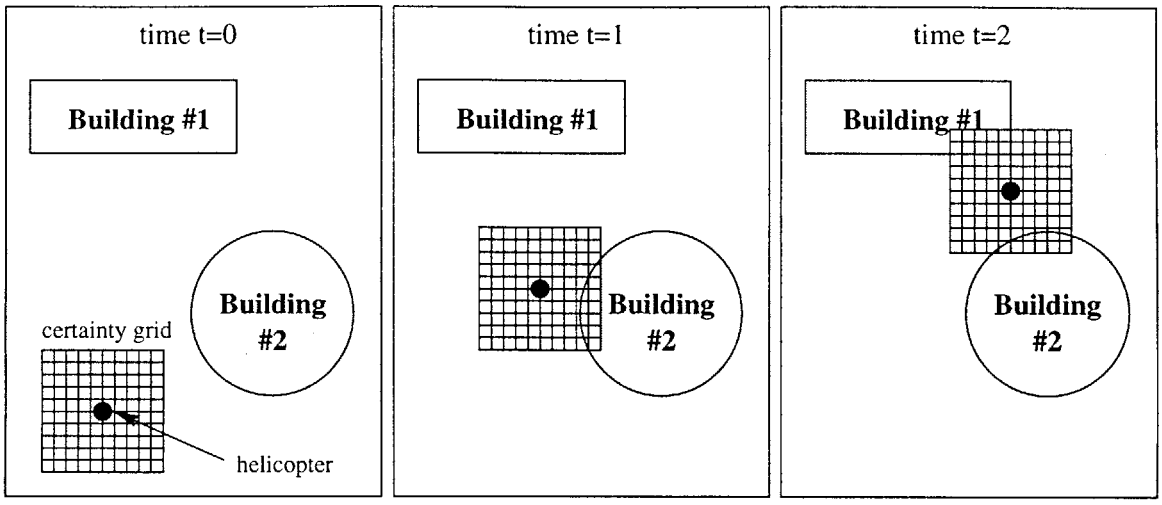

Figure 6-2: Movement of the local map over time

local map. As the helicopter travels through a region, the (x,y,z) position of the certainty grid will change, since the certainty grid moves with the helicopter.

The local map is designed to have the autonomous helicopter approximately centered in the certainty grid. The user chooses the offset of the helicopter in the certainty grid by setting the `x_desired_offset`, `y_desired_offset`, and `z_desired_offset` variables. Typically this offset location will be in the center of the certainty grid, but any location in the certainty grid is permissible. The local map is then positioned so that the helicopter lies close to this offset location. While the exact position of the helicopter in the cell may vary (see Section 6.3), the helicopter will always be located somewhere near this selected offset location.

It is not necessary for the helicopter to be in the center of the local map, and for some possible sensor implementations, it would be better to not have the helicopter centered in the local map. For example, when the scanning laser rangefinder is oriented to look downwards towards the ground, it would be advantageous to offset the helicopter to a position near the top of the local map. This would allow for more cells to be placed in the lower portion of the map, where range readings are being taken. Since there would be no range measurements made for the region above the autonomous helicopter, there would be less motivation to maintain a detailed map of this region.

6.2 Adding Data to the Map

6.2.1 Making Sense of the Data

The primary purpose of the local map is to provide fast, reliable integration of the data provided by the scanning laser rangefinder. As described in Chapter 2, the scanning laser rangefinder sends a stream of range and angle samples to the autonomous helicopter's on-board computer. In the current implementation of the scanning laser rangefinder, the on-board computer receives (range, angle) pairs at a rate of 500Hz. In the following discussion, a (range, angle) pair is referred to as a sample.

The samples from the scanning laser rangefinder are buffered by the on-board computer until the next update of the local map. In the current on-board system, the local map is updated at a rate of 20 Hz along with the rest of the autonomous helicopter's on-board software. Therefore every update of the local map incorporates 25 new samples from the scanning laser rangefinder. This number may change in future implementations of the mapping system, which are discussed in Chapter 9.

The first step in adding scanning laser rangefinder data to the local map is to reject samples that are obviously incorrect. For example, if the provided range is less than the stated minimum range of the scanning laser rangefinder, then this sample is an error and is discarded. On the other hand, if the range is greater than the stated maximum range of the scanning laser rangefinder, then it is assumed that the scanning laser rangefinder did not sense any obstacle along the ray specified by the corresponding angle. This case is flagged as a miss and incorporated into the local map. Finally, if the measured angle is well outside of the laser rangefinder's stated field of view, then this sample is also an error and is discarded.

There are two valid types of samples, hits and misses. A hit is any sample that falls within the maximum range of the scanning laser rangefinder, and represents the range to a detected obstacle. A miss is any sample that extends beyond the maximum range of the scanning laser rangefinder, which represents a line segment along which there are no obstacles.

For samples that are hits, the line segment connecting the scanning laser rangefinder to a nearby obstacle provides two separate pieces of information to the autonomous helicopter's on-board computer. First, the end point of the line segment represents the location of a reflective obstacle. Thus the location of this obstacle should be marked as occupied in the autonomous helicopter's certainty grid. Second, the the existence of a line segment from the helicopter to an obstacle indicates that there is a region of unoccupied space between the helicopter and the obstacle. This region

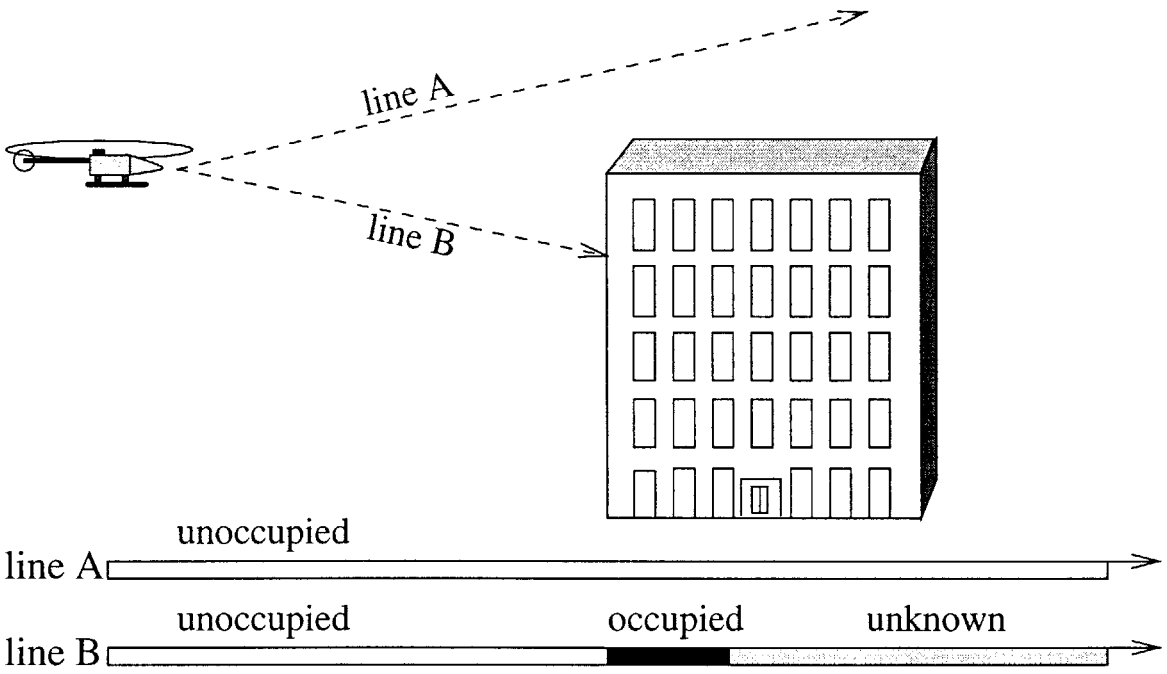

Figure 6-3: Scanning laser rangefinder hits and misses

must be unoccupied because otherwise the scanning laser rangefinder would have registered a hit on this closer object instead.[2] This region should in turn be marked as unoccupied when it is incorporated into the local map.

Samples that are misses are integrated into the local map as line segments extending to the maximum range of the scanning laser rangefinder along which there are no obstacles. The region represented by this line segment is marked as unoccupied in the local map.

Figure 6-3 shows two possible laser rangefinder measurements and their interpretations in the certainty grid. Along line A, the laser rangefinder does hit anything all, so the maximum range value sent to the mapping software. This indicates to the local map that the entire length of line A is unoccupied, as shown on the bar at the bottom of Figure 6-3. Along line B, the scanning laser rangefinder registers a hit. This means that line B is unoccupied in the region between the helicopter and the building, occupied at wall of the building, and unknown behind this. Nothing is known about the portion of line B that extends beyond the wall of the building because the scanning laser rangefinder can't see this (the building is in the way!).

[2]For clarity, this section does not discuss the possibilities of transparent obstacles and reflections. Chapter 9 discusses the effects errors in the scanning laser rangefinder data

6.2.2 Marking a Line in the Certainty Grid

For the purposes of marking cells in the certainty grid, the laser is assumed to be a one-dimensional ray of light. It is assumed that the laser beam does not fan out at all, so the path from the scanning laser rangefinder to the obstacle is a line segment with zero width.[3] This line segment will pass through a number of certainty grid cells along its path from the scanning laser rangefinder to the obstacle grid. Each cell that the line segment passes through should be marked accordingly as occupied or unoccupied in the local map. Thus the line segment discussed in the previous section needs to be discretized into a set of component cells.

Discretization of line segments is a problem that has been rigorously investigated in the study of computer graphics. In order to display a line segment on a computer screen, the line segment needs to be broken up into pixels on the computer screen. One method of discretizing a line that is commonly used in computer graphics is Bresenham's Algorithm. Bresenham's Algorithm is quick, efficient, and easily implemented in three dimensions.[10]

Using Bresenham's line-drawing method, local map cells along the laser line segment are selected and marked as unoccupied. At the end of the laser line segment, the final cell is marked as occupied if the sample is a hit, or unoccupied if the sample is a miss.

If the line segment extends beyond the boundaries of the certainty grid, it is clipped using a simplified version of the Cohen-Sutherland line-clipping algorithm. This is done to minimized the computation associated with line drawing, since the simplified Cohen-Sutherland algorithm is more efficient than checking if each point on the line is within the boundaries (see Appendix C). The Cohen-Sutherland algorithm is a standard method for clipping lines to two or three-dimensions, and is described in detail in [10].

Figure 6-4 shows the result of single laser rangefinder hit in a two-dimensional certainty grid. As before, the black cells represent occupied regions, the white cells represent unoccupied regions, and the gray cells represent regions of unknown occupancy. Along the length of the laser path (marked by the diagonal line), cells are marked as unoccupied, since it is known that the laser did not hit anything along this portion of the line segment. The final cell in the line segment is marked as occupied, since there must be an object at this point to reflect the laser. Of course the rest of

[3]Interestingly enough, the following discretization is still valid even if the laser beam is not at all one-dimensional. In [4], Borenstein and Koren showed that it is possible to obtain a useful and accurate certainty grid when making an assumption of one-dimensionality for Polaroid sonar sensor. Polaroid sonar sensors are known to have a field of view of approximately 30°.

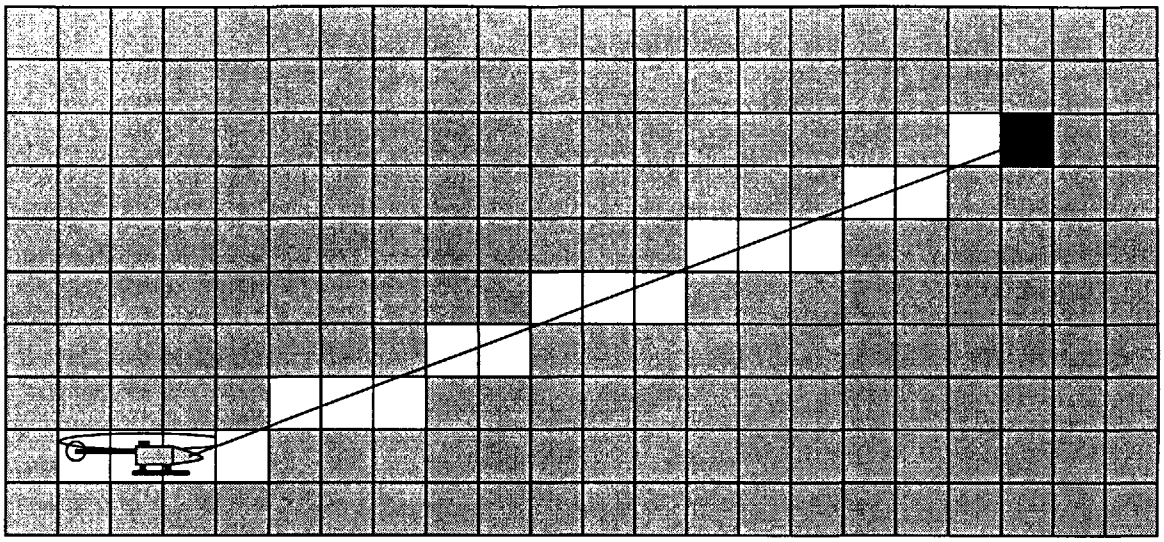

Figure 6-4: Marking a line in the certainty grid

the certainty grid remains unknown, since the laser rangefinder has not yet scanned this area.

6.2.3 Uncertainty Modeling

Each cell in the certainty grid stores an 4-byte integer representing the "occupiedness" of the volume represented by this cell.[4] This number is known as an "occupancy value." All cells are initialized to zero. Every time a cell is marked as occupied, its occupancy value is incremented by one. Every time a cell is marked as unoccupied, the value is decremented by one. This means that the integer stored in the cell will be equal to the net sum of hits and misses for that cell. A cell with a large, positive occupancy value therefore has a high probability of being occupied. Conversely, a cell with a large, negative value is very likely unoccupied. Occupancy numbers with smaller absolute values indicate decreasing degrees of uncertainty. An occupancy value of zero means that there is not enough information to determine whether a cell is occupied or unoccupied.

These simple formulas for incrementing the occupiedness of local map cells are roughly equivalent to other, more complicated formulations of occupiedness. Consider for example the following, more rigorous formulation of occupiedness, used in many other certainty grids [15][5]: Each cell stores a floating point value equivalent to

[4]In this discussion, "occupiedness" is defined as "a numerical measure of the certainty that a region is occupied (as opposed to unoccupied)."

the percentage chance that a cell is occupied. In [8] it is shown that using one floating point number to represent the occupiedness of a cells is mathematically equivalent to using two numbers to represent the portion of the cell that is occupied and the certainty of occupation. The following discussion shows that under certain circumstances, the proposed integer occupiedness representation is equivalent to using one floating point occupiedness value.

Thus in a typical certainty grid, cell values will be floating point numbers between 0.0 (0%) and 1.0(100%), with an initial value of 0.5 (50%) indicating no information about a cell. A cell storing the value 0 (0%) is definitely unoccupied. A cell storing the value 1 (100%) is definitely occupied. A cell storing the value 0.5 (50%) is unknown, which can occur either because no information has been received about this cell, or because there are equivalent amounts of information for and against the cell being occupied.

In the typical certainty grid formulation, cells are marked as occupied according to the equation,

$$x_{t+1} = c_{y_t} * y_t + (1 - c_{y_t}) * x_t$$

where x_t is the occupiedness value stored for a given cell, y_t is the new occupiedness measurement (also in the range $0.0 < y_t < 1.0$), and c_{y_t} is the confidence in the new occupiedness measurement. There is a similar equation for marking cells as unoccupied. In the scanning laser rangefinder, y_t is constant over time since marking a cell occupied at t_1 is the same as marking a cell occupied at t_2. Also, c_{y_t} is constant over time, since there is presently no model for increasing or decreasing the confidence in the scanning laser rangefinder data. Using these assumptions, the following substitutions can be made,

$$d = c_{y_t} * y_t$$

$$e = 1 - c_{y_t}$$

where d and e are constant. Using these substitutions, Equation 6.2.3 can be simplified to,

$$x_{t+1} = d + e * x_t$$

which has an additive increase for occupiedness just like the integer update rules presented earlier. The only remaining difference between this version and the integer version is the scaling to place the occupiedness value on the range (-inf, inf) as opposed to (0.0, 1.0). This can be done using the substitution

$$x_{integer} = 1/x_{float} + \frac{1}{x_{float} - 1}$$

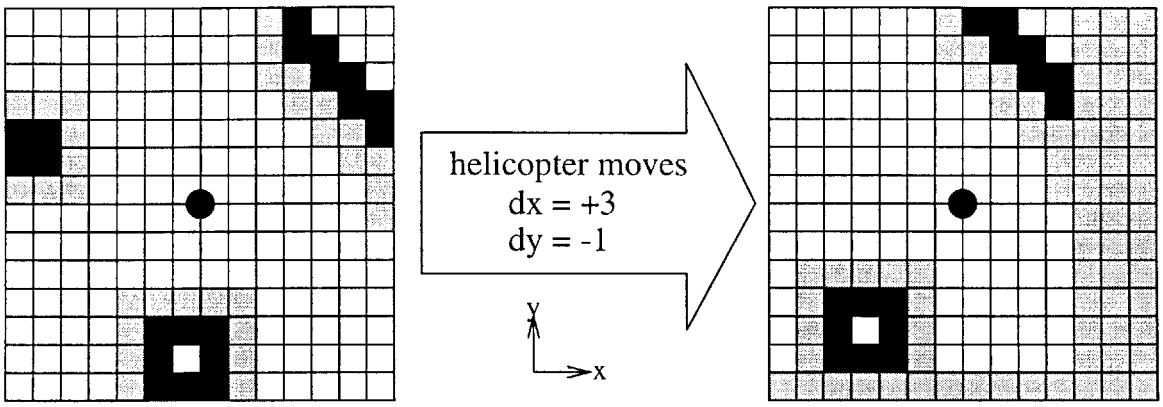

Figure 6-5: Shifting the Local Map

Thus in the case of the autonomous helicopter, the simple integer occupiedness representation is equivalent to a more complicated floating point representation.

Using the simple integer formulation for occupiedness also has other advantages. Most computers will perform the single integer addition operation more quickly than the two multiplications and two additions that are necessary for the floating point variation. Since these operations are performed 500 times per second, using the integer formulation helps to speed up the on-board software. Using integers also helps to reduce space that the local map occupies in memory. In ANSI C, a variable of type `float` takes up 4 bytes in memory, whereas a `shortint` type requires only 2 bytes. Minimizing the memory size of the local map will in turn help to speed up on-board computer by making more memory available for other purposes. Finally, the integer formulation also has the advantage of simplicity. It is simple to understand, simple to debug, and will allow maximum flexibility of interpretation by programs making use of the local map.

6.3 Moving the Map

The local map is maintained relative to the autonomous helicopter, with the autonomous helicopter always near a specified offset location in the certainty grid (usually at the center of the grid). As the helicopter moves, the local map must be shifted so that objects in the map maintain their positions relative to the helicopter.

A two-dimensional example of shifting the map is shown Figure 6-5. In this diagram, the helicopter moves a distance of +3 cells in the x dimension and -1 cell in the y dimension. Since the map moves with the helicopter, all cells in the map are shifted by in the opposite direction. Notice how the square block near the bottom of

the local map is shifted to the left (negative x) and upwards (positive y) in the second certainty grid. Also, the right hand side of the certainty grid has 3 unknown rows because this region is new to the map and has not yet been scanned by the helicopter.

Two issues need to be addressed when shifting the local map certainty grid according to helicopter movements. First, what happens when the displacement of the helicopter is a non-integer number of cells? In order to maintain the local map centered directly around the helicopter, the certainty grid from time $t-1$ would have to be interpolated in some way to be copied into the certainty grid for time t. This would be necessary because the cells from time $t-1$ would no longer line up evenly with cells at time t. No matter how well the interpolation is done, this method will always result in some blurring of cells values and obstacle positions.

The second issue, which is closely related to the first, is how to handle small perturbations in the helicopter position. Even when the autonomous helicopter is attempting to hover in one position, there will always be small variations in the helicopter position because of vibration, wind gusts, servo position fluctuations, and a host of other factors. One way to handle these variations would be to shift the certainty grid accordingly at each time step. However this would result in unnecessary blurring of the local map, since these small shifts in the local map would require interpolations. Just like in the first issue, interpolation would need to be performed on certainty grid cells from time $t-1$ that do no line up evenly with certainty grid cells at time t. This interpolation issue needs to be resolved in order to produce a useful and precise local map.

The local map is shifted only in cell length increments. This is done to simplify and streamline the shifting process, and to minimize the rounding errors associated with shifting a certainty grid. However, shifting the certainty grid in cell length increments means that the autonomous helicopter will not stay at the exact desired offset position in the certainty grid. Instead, the exact position of the helicopter in the local map is allowed to deviate slightly from the desired offset position. This makes it easier to shift the local map since shifting only needs to be done in integer increments, and minimizes the shifting necessary to mimic small variations in the helicopter's position.

Ideally, the local map would be shifted every time the helicopter moves a distance of at least one cell length. However, shifting takes a lot of time,[5] so for speed reasons it is better to shift the local map only after the helicopter has moved a greater distance

[5]In order to shift the local map, each cell must be copied into from the old certainty grid into the new, shifted certainty grid. If the certainty grid contains hundreds of cells, this can take a long time.

from its original location. The variable `max_deviation` sets this maximum amount of deviation from the chosen helicopter offset in the local map.

The local map maintains two measurements of the position of the helicopter in the local map. The first of these is the `x_desired_offset`, `y_desired_offset`, and `z_desired_offset` variables, which are set by the user to be the *desired* helicopter offset. Whenever the helicopter moves, the local map certainty grid will be shifted so that the helicopter will fall somewhere near this specified offset position. The second measurement is the *actual* offset of the helicopter in the local map and is stored in the variables `x_offset`, `y_offset`, and `z_offset`. The local map will be shifted whenever the helicopter position (`x_offset, y_offset, z_offset`) would be more than `max_deviation` away from (`x_desired_offset, y_desired_offset, z_desired_offset`).

Shifting the local map in this way offers many advantages over performing some sort of interpolation of cell values. Since the certainty grid only shifts by integer multiples of cell lengths, shifting is done quickly and easily. Also, since the helicopter's offset position is allowed to vary, it is very easy for the helicopter to hover in one position and use the scanning laser rangefinder to make measurements of the surrounding area. Small perturbations in the helicopter position will not necessitate shifting the local map, and the local map should stay approximately centered on the autonomous helicopter at all times.

6.4 Conclusion

Figure 6-6 shows an overview of the local map update process. First, the local map update function obtains range and angle information from the scanning laser rangefinder hardware. Immediately afterwards, it notes the position of the autonomous helicopter as estimated by the the DSAAV's on-board navigation code. If necessary, the local map is shifted so as to keep the helicopter at the desired offset position. Finally, the new range information is incorporated by drawing a line in the local map certainty grid.

This chapter has provided a comprehensive overview of the functionality of the local map portion of the mapping system. However some details of the computer code have been omitted for the sake of clarity. For example, many of the diagrams have been drawn for a two-dimensional map implementation, when true local map is of course three-dimensional. You may refer to the code in Appendices C and D for further details. Performance of the local map, as well as the mapping system as a

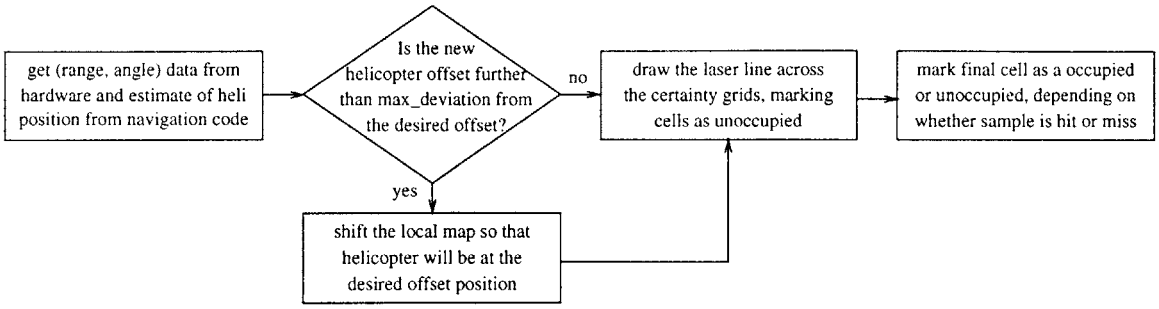

Figure 6-6: Updating the Local Map

whole, is discussed in Section 8.2.

Chapter 7

The Global Map

In addition to the local map certainty grid that is maintained for the region immediately surrounding the helicopter, the mapping system also includes a "global map" that is useful for recording obstacle positions over the course of a mission. The global map is designed to to be an accurate, memory-efficient representation of the obstacles that have been marked in the local map. It will likely be used for mapping large areas, long distance path planning, and multiple-vehicle coordination.

7.1 The Map Structure

The global map consists of a 3D kd-tree in which splitting is done along the longest cell dimension.[1] Like the local map, the global map is oriented in the north-east-down frame of reference used by the helicopter navigation system. Cells from the local map certainty grid are periodically imported and added to the global mapping structure. Cells in the global map with equal or similar "occupiedness"[2] are grouped together into larger cells, thereby minimizing the global map's space in memory. Over time, the global map will produce a minimally subdivided kd-tree representation of the environment sampled by the scanning laser rangefinder. The code for the global mapping structure is included in Appendices C and D.

Like quadtrees, the top level cell of kd-trees should encompass the entire region that is being mapped. For the global map, the base cell's extent should be initialized to be large enough to include the entire volume that will be sampled by the scanning laser rangefinder over the course of the helicopter's mission. This volume may be as large as many square miles, or as small as a single city block, and can have any desired proportions. Typically the global map base cell will be much longer and wider

[1] kd-trees are introduced in Section 4.3.1.

[2] "occupiedness" is introduced in Section 4.1.2 and explained further in Section 6.2.3.

than it is tall. The base cell is initialized to have occupiedness of zero, no split, and no children.

When subdivision is necessary, cells in the kd-tree are split in half across their longest dimension. For example, a kd-tree cell that is 10 ft long in the x dimension, 20 ft long in the y dimension, and 30 ft long in the z dimension will be split along the z dimension into two cells that are 10ft x 20ft x 15ft. Since all splits are done with planes perpendicular to the coordinate axes, all cells will be axial boxes.

The kd-tree is implemented as dynamically allocated binary tree. Figure 7-1 shows the C code for the structure of a global map cell. All cells in the tree are oriented in the north-east-down frame of reference and must be axial boxes, but there are no restrictions on the size or proportions of cells. The cell size is stored in memory by noting the southern-most, western-most, highest corner of the box, and the northern-most, eastern-most, lowest corner of the box. This representation uniquely defines the location and extent of each cell, and also makes it easy to split cells along any dimension. Each cell stores an integer occupancy value, as discussed in Section 6.2.3.[3]

For traversal purposes, each cell also stores the dimension along which it is split (if any), pointers to each of its two children (if any), and a pointer to its parent cell (if any). The pointer to a cell's parent is especially useful when conglomerating cells to achieve space efficiency. All of these pointers are maintained throughout the kd-tree and set to NULL when not being used.

Since occupiedness values for a given region can be stored at multiple levels in the kd-tree, it is important to define the relationships between these occupiedness values. For this kd-tree, if a cell has children, then the occupancy value stored in this cell is irrelevant. The children cells provide the true occupancy data for the volume. According to this rule, only leaf cells[4] in the tree store the actual occupancy information. All parent cells act just as an organization structure.

7.2 Adding Data to the Map

7.2.1 Overview

As the autonomous helicopter flies a mission, the local map filters and records information about surrounding obstacles. The obstacle data stored in the local map needs to be periodically transferred to the global map to maintain currency and consistency

[3]Occupancy values indicate the confidence that a certain cell is occupied. A high positive occupancy value means the cells is most likely occupied with a solid object, whereas a low negative occupancy value indicates that a cell is most likely unoccupied.

[4]In a binary tree, any cell that has no children is known as a leaf.

```
typedef struct global_map_cell {
  int occupied;          /*high=occupied, low=open*/
  float x0;              /*first corner of the cell*/
  float y0;              /*first corner of the cell*/
  float z0;              /*first corner of the cell*/
  float x1;              /*second corner of the cell*/
  float y1;              /*second corner of the cell*/
  float z1;              /*second corner of the cell*/
  enum dimension split;  /*split dimension to create children*/
  struct global_map_cell *parent;
  struct global_map_cell *child0;
  struct global_map_cell *child1;
} global_map_cell;
```

Figure 7-1: Structure of a global map cell

in the global map. Transfers from the local map to the global map happen at a specified update rate, which can be adjusted to be any multiple of the update rate of the local map. While a faster global map update rate is better for currency and consistency, a slower update rate can be used to reduce the amount of processing time that is spent updating the global map.[5] A balance is found based on the performance on-board computer system. Typically the global map will be updated once for every five to ten updates of the local map.

The global map is updated by sequentially copying all the cells from the local map into the global map. For each local map cell, the update process begins by reading in the cell's location, extent, and occupancy value. Since all local map cells are cubes and all global map cells are boxes, the cell types are compatible. However there is no guarantee that the local map cell will align perfectly with the global map cell. The local map cell could be an uneven size, or could be offset from the global map kd-tree cells. Therefore a detailed addition process is necessary.

The addition process is a recursive function that subdivides the global map kd-tree as necessary to add the new cell. Starting at the kd-tree base cell, the function proceeds as shown in Figure 7-2. First, the top level kd-tree cell is split along the y dimension, creating cell A. Next, cell A is split along the x dimension, creating cell

[5]Since the global map perform dynamic memory allocation and deallocation and stores a very large amount of data, it's performance can be slow. See Section 8.2 for further details.

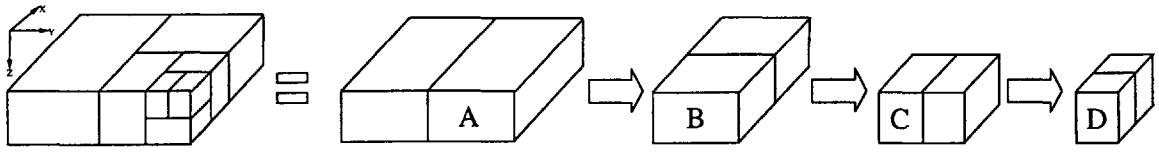

Figure 7-2: Subdivision of the Global Map

B, which is in turn split to create cell C, and so on.

7.2.2 Marking a Kd-tree Cell

Figure 7-3 shows a flowchart for the global map addition process. First, a check is done to see if the cell being added is approximately the same size as the current kd-tree cell. If so, then the function will attempt to mark the current kd-tree cell with the occupancy information from the new cell. If the kd-tree cell is a leaf, then it can be marked by simply adding the occupancy of the new cell to the occupancy of the existing kd-tree cell. This works correctly because the local map and the global map both use the same occupancy representation. Thus each cell in the global map will be consistent with the mission totals of the laser rangefinder information gathered for that region.

On the other hand, if the kd-tree cell has children, then the current kd-tree cell should not be marked. (Remember, only leaf cells store occupancy information.) Instead each of current kd-tree cell's children should be marked with with occupancy information from the new cell. Of course if each of these children has children of its own, then another recursion is necessary before the occupancy values can be added. The marking process proceeds until it has marked all leaf cells that are descendents of the current kd-tree cell.

7.2.3 Splitting the Tree

If the cell being added is smaller than the current kd-tree cell, then the kd-tree cell needs to be subdivided. Subdivision proceeds as follows: If the current kd-tree cell is not yet split, then the memory is allocated for the cells children and these children are linked to the current cell. The dimensions of the two child cells are set by splitting the parent cell along its longest dimension and storing the extents of each half of the region in each of the respective child cells. The first child position always stores the child on the negative side of the split plane and the second child position always stores the child on the positive side of the split plane.

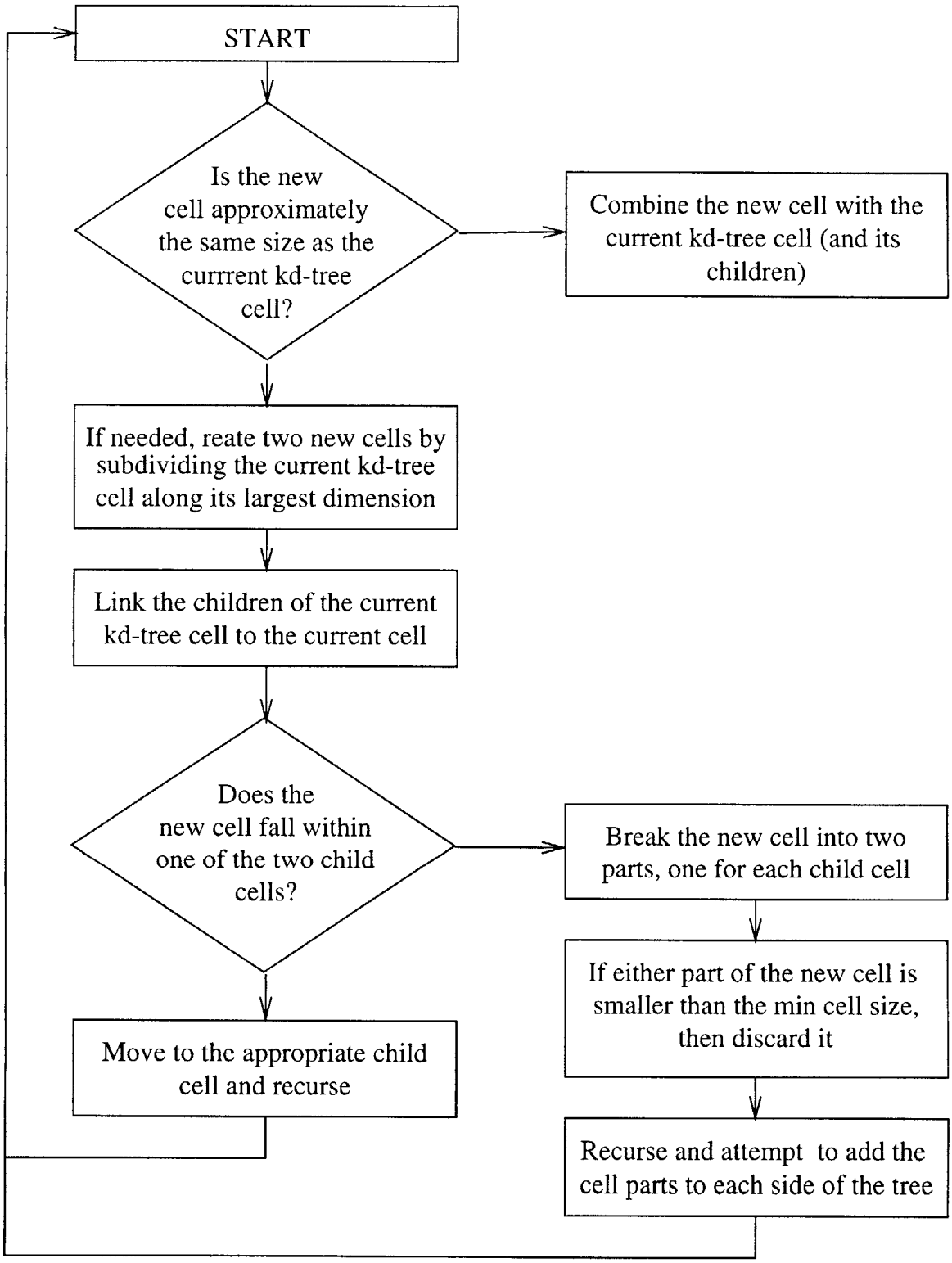

Figure 7-3: The Global Map Addition Process

The kd-tree will continue splitting and generating new children until the current kd-tree cell size is approximately the same as the new cell size, or until the kd-tree cell size reaches a preset minimum size. Setting a minimum cell size prevents the tree from becoming too deep and overly precise. This helps to save on memory and speeds up the cell addition process.

Often the new cell will fall across the split plane for the current kd-tree cell. In this case, the new cell also will be broken into two pieces using the split plane. This produces two smaller new cells than need to be added to each child of the original kd-tree cell. The process will then recurse and attempt to add each of these new cell pieces to the appropriate kd-tree region.

7.3 Conglomeration

After every update of the global map, a conglomeration function is run on the global map to minimize the amount of subdivision. Since minimizing the amount of subdivision also minimizes the number of cells in the global map, conglomeration helps to minimize the memory requirements of the global map.

The basic principle of the conglomeration process is that if both of the children of a parent cell have the same occupancy value, then the children can be deleted and their occupancy value stored in the parent cell. No information is lost because the parent cell represents the same volume that was represented by the two child cells. Memory is saved because the total number of cells in the global map is reduced by two (from the deletion of both child cells).

In order to increase the amount of conglomeration that occurs in the global map kd-tree, the user is allowed to select a conglomeration tolerance. Using the conglomeration tolerance addition, two child cells will be conglomerated if their occupancy values differ an amount less than or equal to the conglomeration tolerance value. For example, if the conglomeration tolerance is set to 3, child A has an occupancy value of 15, and child B has an occupancy value of 17, then child A and child B can be conglomerated because their difference (2) is less than the conglomeration tolerance.

Conglomeration proceeds by traversing the kd-tree to find the leaf nodes and then conglomerating up the tree. The C code to perform this calculation is included along with the other mapping system functions in Appendix C.

7.4 Conclusion

The global map is a straightforward kd-tree representation. It provides an efficient, flexible way of storing volumetric information, and makes a fitting complement for the local map. Many of the strengths of the global map are a direct benefit of the hierarchical kd-tree representation. While it is likely that certain aspects of the mapping system will evolve in future versions of the autonomous helicopter, it is likely that the hierarchical nature of the global map will endure.

Chapter 8

Implementation and Results

8.1 The Simulation Framework

8.1.1 Motivation for Using the Sim

The described mapping system was implemented in the Draper simulation framework. While such an implementation was not absolutely necessary for the demonstration of a prototype system such as this one, it was felt that inclusion of the mapping system in the simulation framework would be necessary for future map-based projects. All of the current on-board software for the autonomous helicopter is implemented in the simulation framework, and the simulation is used for extensive testing of all autonomous helicopter flight algorithms.

The simulation will be particularly useful when the mapping system is used to accomplish the long-term goals of obstacle avoidance, path planning, and landmark-based navigation. Algorithms such as obstacle avoidance require careful tuning of the helicopter flight path, and can not easily be tested without the use of a simulation.

In addition, it was originally thought that use of the simulation framework would help to reduce development time for the mapping system. Unfortunately, issues of integration with the framework overshadowed many of the advantages it provided. However, implementing the mapping system in the simulation did help to decrease the development time of the entire scanning laser rangefinder project by enabling hardware and software development to be performed in parallel. Without a suitable simulation in which to test the mapping system software, testing of the simulation software would have to have been postponed until after the hardware was in perfect working order.

8.1.2 Simulation Implementation

The Draper simulation framework is an extension of the C programming language. It consists of a structural preprocessor and real-time debugger that are designed to streamline the development of navigation code. In order to use the simulation framework, any new software must include a specially-formatted header file that defines all variables and structures for the software. This file must have a `.spec` or `.spech` extension, and is digested by the framework preprocessor to create appropriate `.c` and `.h` files. Only after this preprocessing has been done can the simulation be compiled and run from within the simulation framework. The framework's real-time debugger allows users to view and change variable values on the fly, much like a conventional debugger. Reference [1] provides an overview of programming and running software from within the Draper simulation.

The mapping system software was implemented in simulation framework as part of the autonomous helicopter's on-board code. The on-board code is the code that would be run on the autonomous helicopter's on-board computer during actual missions. Mapping system calls were interspersed with calls to the navigation, guidance, and control software that was already implemented on the DSAAV.

The Draper autonomous helicopter simulation also generates a 3D image in accordance with the simulation. This animated view of the helicopter and its surrounding is useful for viewing the action taking place in the simulation and performing demonstrations of the autonomous helicopter capabilities.

Two other graphical windows were created to display the local map and the global map. These windows complement the graphics of the 3D window by providing an easy way to see the autonomous helicopter's object representations. Using the GL window interface provided by the simulation 3D window and OpenGL graphics commands,[1] 3D grids were drawn and colored to represent the cells in the local and global maps. As information is obtained from the simulated scanning laser rangefinder, the graphical map representations are updated accordingly to show what the autonomous helicopter "sees."

Figure 8-1 shows a screen capture from the Draper simulation implementation of the mapping system. In the top left pane of Figure 8-1 is the simulation command window, which is used for entering all simulation commands. The bottom left pane shows the simulation view window (also known as the 3D window). This window

[1] Draper is currently working on converting all the graphics in the simulation from GL to OpenGL. Thus it was important to write all new graphics code in OpenGL, even though not all OpenGL commands work in the hybrid windowing system that currently exists.

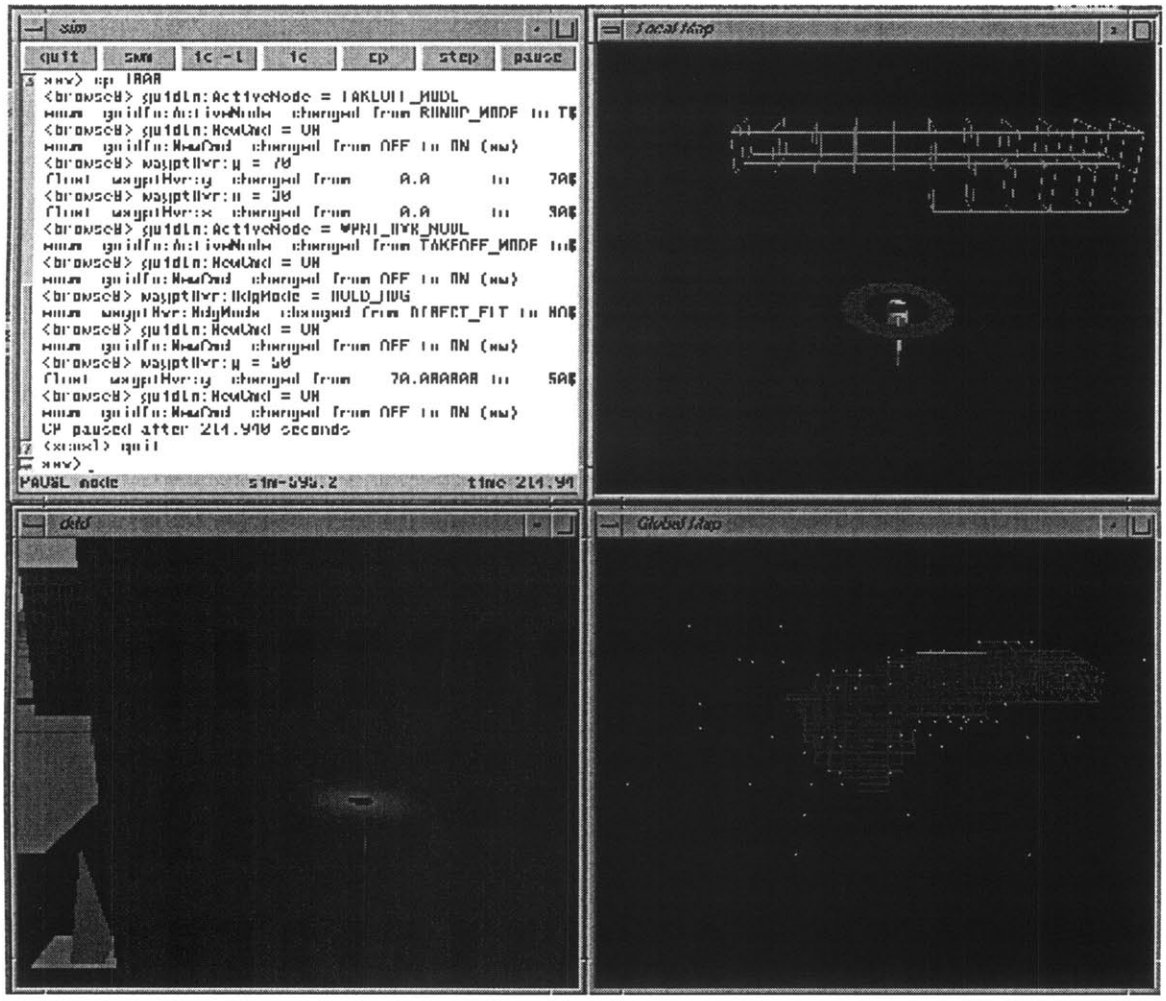

Figure 8-1: The Simulation Graphics Windows

displays an image of the autonomous helicopter flying through the simulation world. In the screen capture shown in Figure 8-1, the helicopter is right in front of an office building.

The two right-hand windows in Figure 8-1 are local and global map windows. The local map window (the top right pane of the figure) indicates that the local map is storing a set of occupied cells directly in front of the helicopter. These cells correspond to the office building shown in the 3D window. Notice that since the laser rangefinder only scans in the horizontal dimension, the local map only represents a horizontal cross-section of the building, and that there is no data about the back side of the building. Finally, the global map window (the lower right pane of the figure) shows the conglomerated path of the helicopter since the start of the mission.

8.2 Performance

8.2.1 Local Map Performance

The local map performs according to its qualitative specifications, receiving data from the simulated scanning laser rangefinder and filtering this to produce estimates of obstacle positions. While it is difficult to give quantitative measurements of the local map performance, the following paragraphs give a brief overview of some qualitative aspects of the local map operation.

The certainty grid filters data from the scanning laser rangefinder very well. Even when the simulated rangefinder is set to produce many errors, the local map still produces a useful map of approximate obstacle positions. It is likely that this robustness is a result of the large number of samples provided by the scanning laser rangefinder and the inherent filtering properties of a certainty grid representation.

Figure 8-2 shows a MATLAB plot of the certainty grid occupancy values that result from numerous samples of the range to a static object. In this figure, the x axis represents the range along a straight line away from the helicopter, and the y axis represents the occupancy value stored in each cell along this line. The obstacle is at location $x = 30$, and the laser rangefinder errors are modeled as a Gaussian distribution with standard deviation 3.0. In this simulation, a miss increments a cell by -1 and a hit increments a cells by +10. Notice how the region between $x = 0$ (where the helicopter is) and $x = 27$ (in front of the obstacle) is negative, indicating that it is unoccupied, and the region surrounding the obstacle itself is positive, indicating that it is occupied. The region behind the obstacle ($x > 35$) has an occupancy value of approximately zero, since no data has been collected about this region.

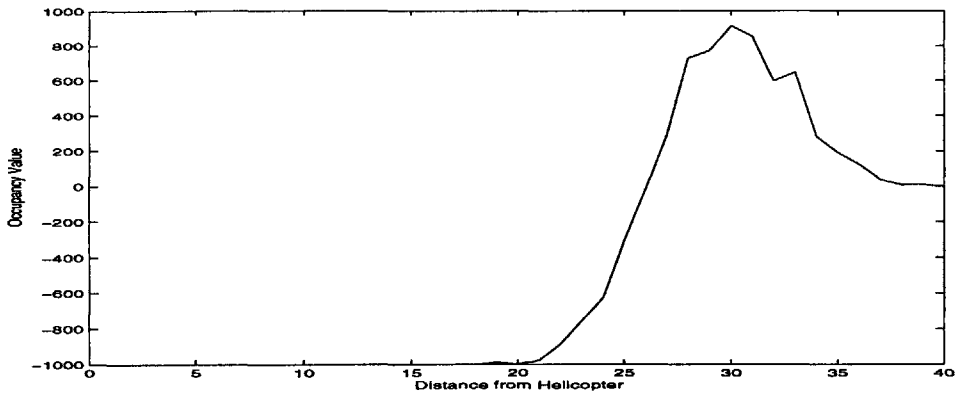

Figure 8-2: Filtering in the Certainty Grid

Figure 8-3 a screen capture that illustrates the error-filtering capabilities of the mapping system. In this test run, the simulated laser rangefinder has been configured to produce errors in a Gaussian distribution with a standard deviation of 10 feet. The lower right frame of this figure is the 3D simulation window, which shows the DSAAV positioned in front of an office building. The top frame shows the local map, in which occupied cells are represented by solid cubes, and unoccupied cells are represented in wire frames. The lower right frame shows the global map, in which occupied cells are drawn as wire frames. In order to show the error-filtering capabilities of the mapping system, a wireframe representation of the building is also drawn in this frame. Even with the large errors in the scanning laser rangefinder data, the map is still able to note the solid wall of the office building.

Implementing the local map certainty grid as an array of integers seems to have worked well in minimizing the access time and memory requirements of the representation. Accessing the data is quick enough that it does not slow down the update process, and because of the large number of errors and great quantity of scanning laser rangefinder data, there is little need for a more complicated certainty representation.

However, as the number of cells in the local map grows, there is noticeable slowdown in the mapping system performance. This is mostly caused by the need to update more cells at each call to the mapping system. Even though the local map certainty grid has been designed to minimize cell access time, the computer can be overwhelmed by the sheer number of cell accesses necessary to maintain a high resolution certainty grid. Noticeable slowdown occurs in local map certainty grids with over one thousand cells.

One pleasant surprise was that the roll and tilt of the autonomous helicopter vary

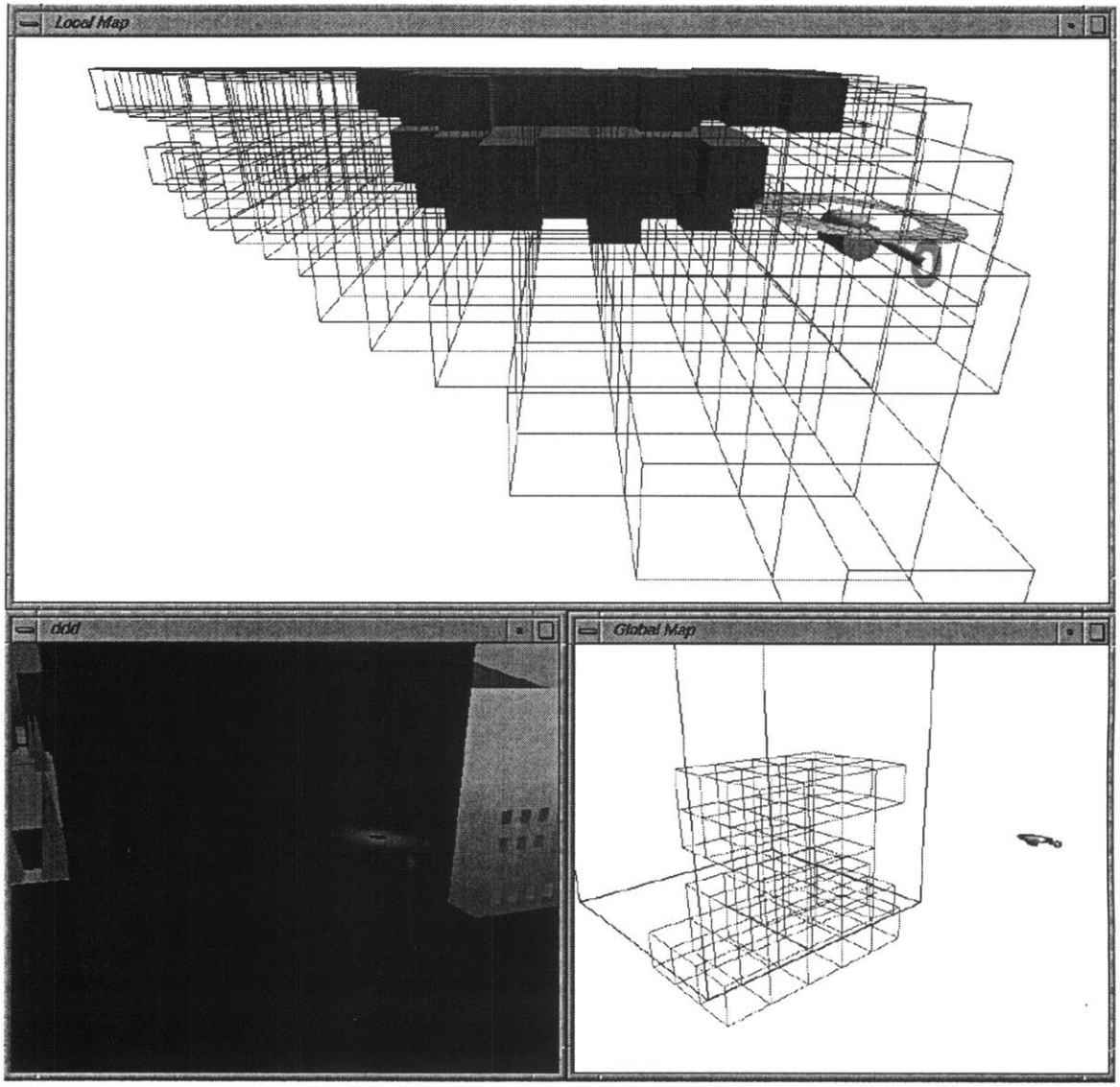

Figure 8-3: Errors in the Certainty Grid

only minimally during simulation test flights. During a series simulated missions, the autonomous helicopter's typical roll and tilt deflections were only a few degrees. This is significant because the scanning laser rangefinder is designed to be mounted facing forward on the autonomous helicopter and only scans horizontally. Thus the scanning laser rangefinder typically only marks cells in same horizontal plane as the autonomous helicopter. Cells above and below this horizontal plane are left completely unmarked unless the autonomous helicopter moves up or down, in which case the shifting of the local map will modify their values.

Since the local map cells above and below the autonomous helicopter are almost completely unused, this means that it is possible to reduce the extent of the local map in the z dimension. A little experimentation showed that the local map was able to function perfectly well with an extent of just 3 cells in the z dimension. It was impractical to reduce the z extent of the local map further than this because of variations in the position and orientation of the autonomous helicopter. If the helicopter position varies too much relative to the local map size, not enough filtering will be done before cells are shifted out of the boundaries of the local map. If the helicopter orientation varies too much relative to the map size, then laser scan lines will fall outside of the local map and will remain unmarked. Reducing the number of cells in the local map in turn helped to speed up the update process.

8.2.2 Global Map Performance

The global map produced an efficient and reliable map of the regions scanned by the autonomous helicopter. Again, it is difficult to make quantitative measures of the performance of the mapping system, so the following paragraphs note some qualitative aspects of the mapping system.

The best way to measure the efficiency performance of the global map is according to the amount of memory used. Simulation testing was done by running the autonomous helicopter on quick 2-3 minute missions through an urban environment with appropriate values chosen for the global map size (300ft x 300ft x 100ft), minimum cell size(10ft), and conglomeration factor(10). By the end of these missions, the global map kd-tree typically contained nearly 1000 cells. Since each global map cell stores approximately 50 bytes of data, this means that the global map took up nearly 50 MB of memory.

This representation is reasonably efficient, but not as efficient as might have been possible. One reason for the efficiency performance of the local map is the lack of conglomeration that occurred. The current implementation of the scanning laser

rangefinder only scans horizontally, so it only obtains occupancy information for a horizontal plane at the same height as the helicopter. Since the current DSAAV flight plans keep the helicopter at a constant height throughout the mission, the global map only contains data at one height. It is difficult to conglomerate the cells in this horizontal plane because of the lack of information about cells above and below this plane.

Updating the global map is a time-consuming process, since each cell in the local map needs to be separately added to the global map. This can take time, since updating the global map often will require allocating memory for new global map cells. Also, this process is closely linked with conglomeration of the global map, which also takes time. Conglomeration takes lots of time because it requires traversing the entire global map and freeing up memory from redundant cells. However, it was possible to alleviate this speed concern by reducing the update rate of the global map. Reducing the update rate to 1 Hz had little or no effect on the mapping system performance, and helped to minimize computation speed problems.

8.2.3 Example Test Flight

The mapping system was tested in the urban environment model of the Draper aav simulation. In this simulation, the DSAAV is programmed to fly through a simulated city block, complete with office buildings, bunkers, and warehouses. Most of these buildings can be sensed by the simulated version of the scanning laser rangefinder, which generates data for the mapping system. The urban environment provides a good testing ground for the mapping system because it offers a variety of stationary solid objects to be represented in the mapping system. Mapping these objects demonstrates the capability of the mapping system to represent solid objects, open areas, and inconclusive data. Figure 8-4 shows an overhead view of the urban setting. Notice that some of the office buildings have been positioned at odds angles to produce more interesting results for the mapping system.

Figures 8-5 through 8-11 show a series of screen captures from a test run in the urban environment DSAAV simulation. Figure 8-5 shows the initial states of the local and global maps, immediately after the initialization of the navigation code. Notice how there is only one sweep's worth of samples in both maps. Figure 8-6 shows the contents of the local and global maps immediately after takeoff, when the helicopter is hovering at a height of 30 ft in front of the first building. Notice how the local map only stores cells at the same height as the helicopter since it only maintains 3 cells in the z dimension. In contrast, the global map stores the entire mapping history,

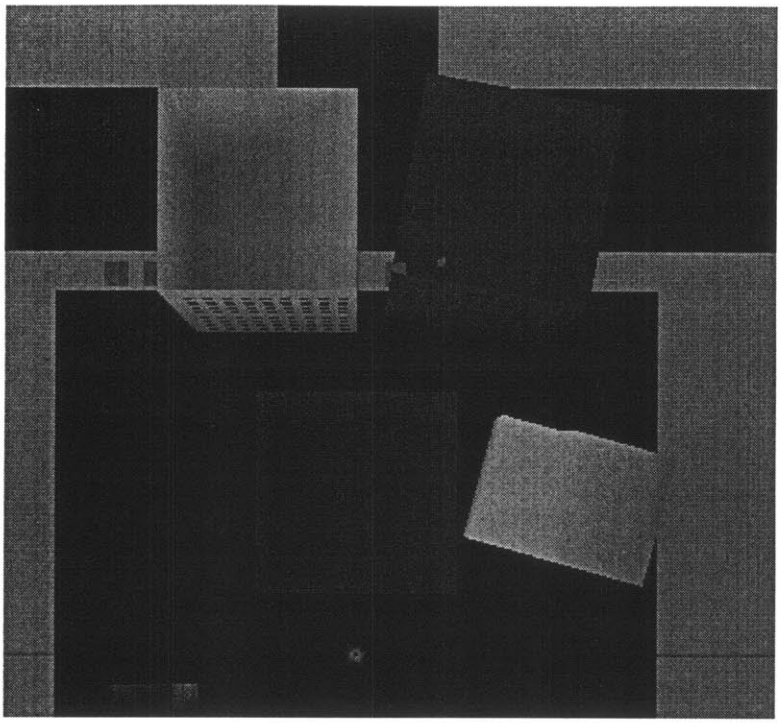

Figure 8-4: Overhead View of Urban Environment

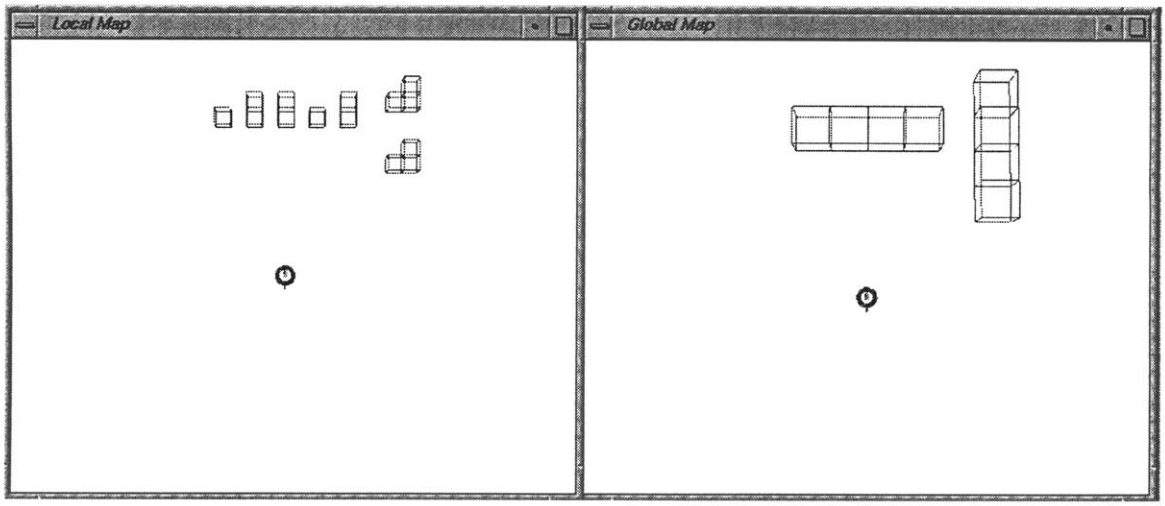

Figure 8-5: Simulation Map Windows

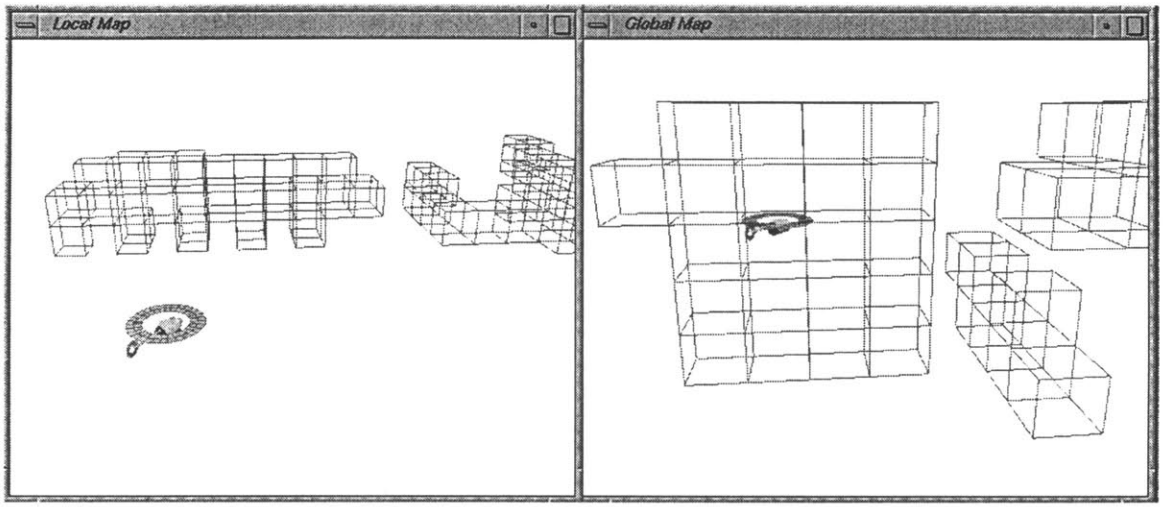

Figure 8-6: Simulation Map Windows

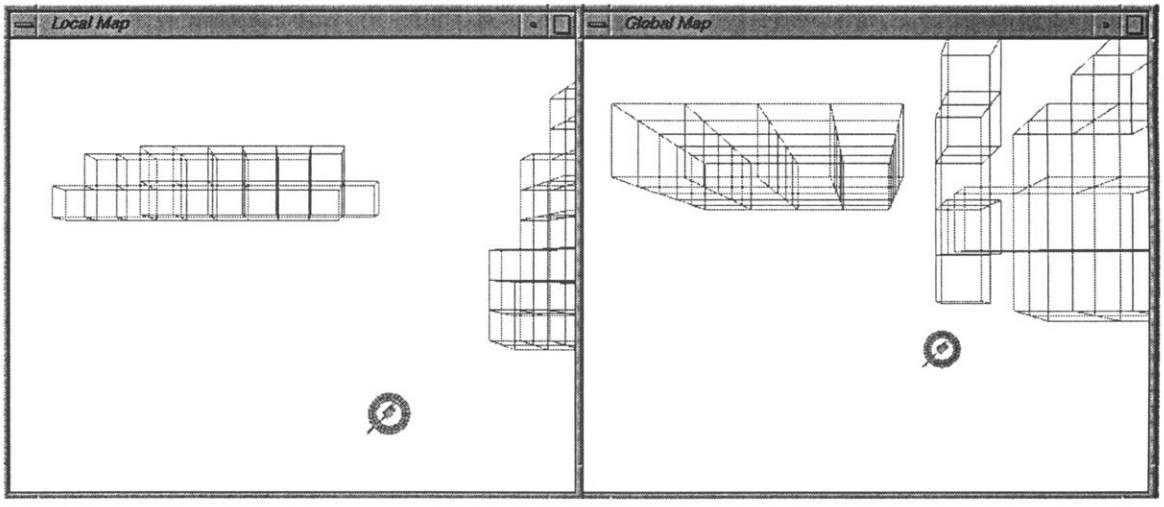

Figure 8-7: Simulation Map Windows

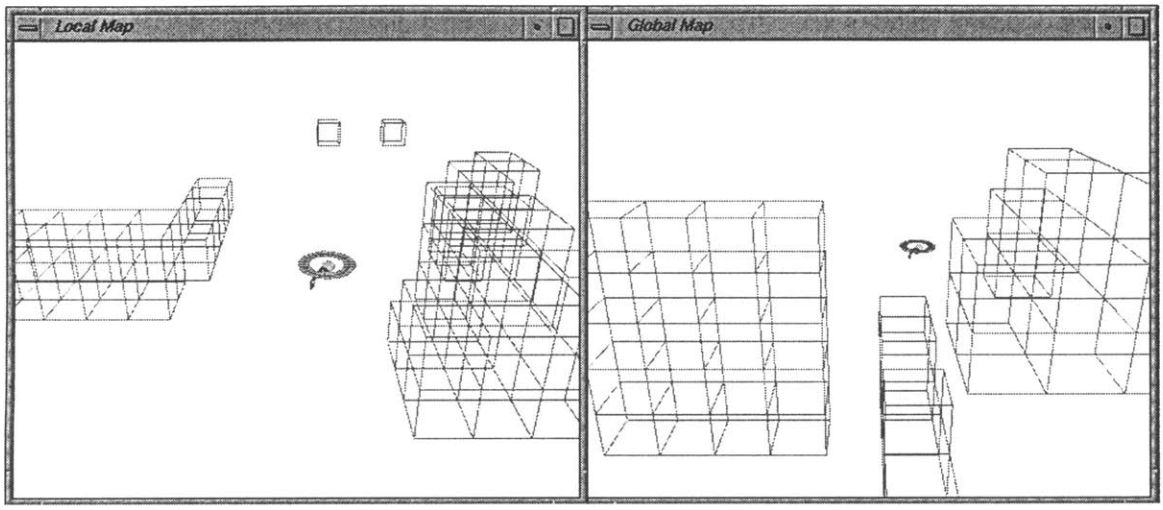

Figure 8-8: Simulation Map Windows

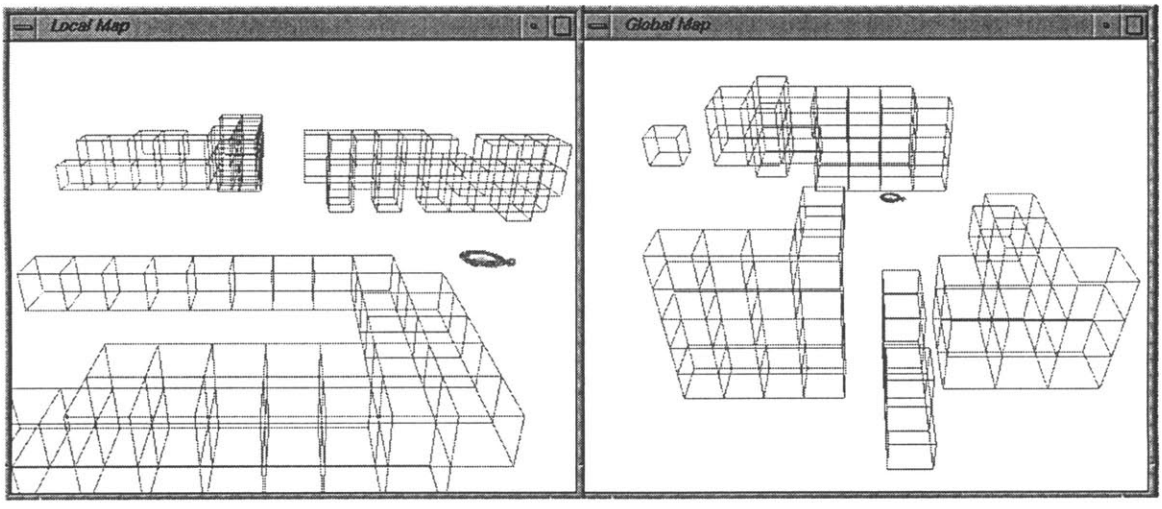

Figure 8-9: Simulation Map Windows

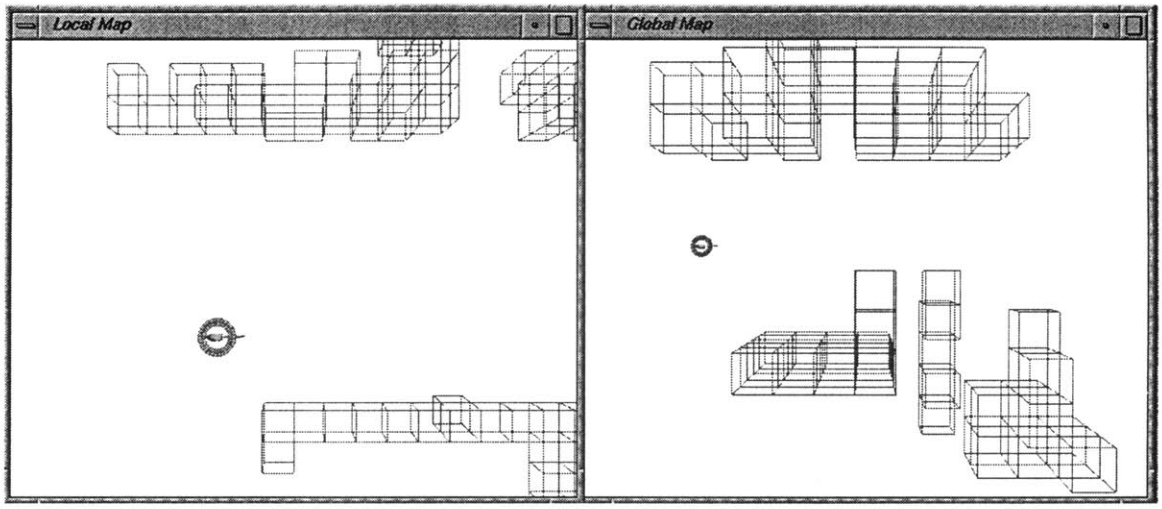

Figure 8-10: Simulation Map Windows

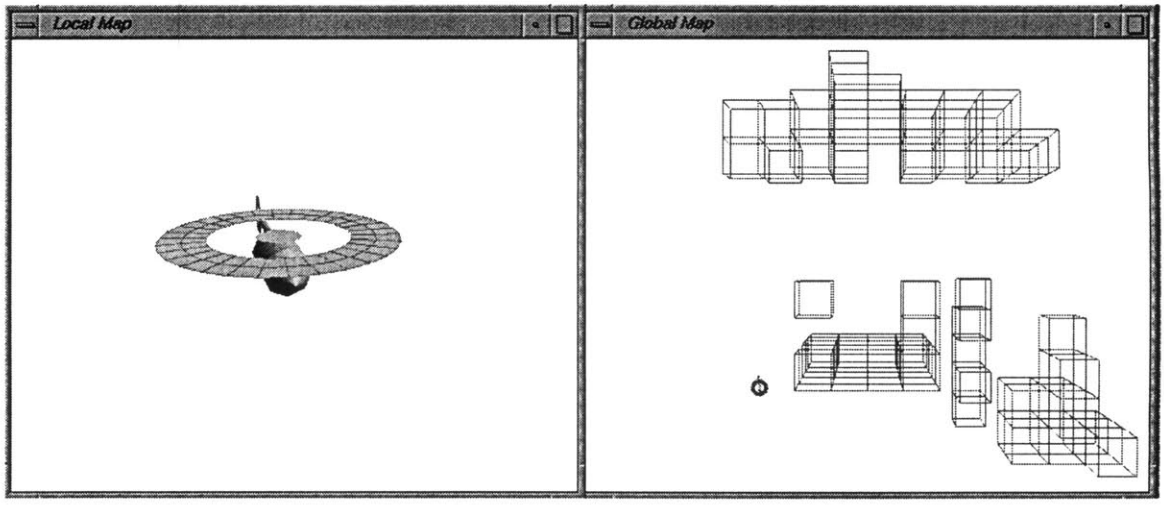

Figure 8-11: Simulation Map Windows

so it is possible to see the entire wall of the building in the global map. Next, the helicopter moves around to the right (the east) of the building, as shown in Figure 8-7, which shows overhead views of both the local and global maps. As the helicopter flies between the two buildings, it passes over the wing of the second building, as is evident in the global map of Figure 8-8. In Figure 8-9, the autonomous helicopter has moved into the courtyard area between the buildings, and the mapping structure is evident in both the local and global maps. Figures 8-10 and 8-11 show the completion of the helicopter flight path, as it finished circling the first building and then lands on the ground. Notice how there are no obstacles in the local map of Figure 8-11, since nothing is being detected and all of the previous obstacles have shifted off the map.

Chapter 9

Conclusion

9.1 Future Work

9.1.1 Local Map Improvements

Since the autonomous helicopter seldom flies at angles with large degrees of roll and tilt and only scans in the horizontal xy plane, it may be possible to completely eliminate the z dimension in the local map. This would greatly reduce the memory requirements of the local map, and should also help to reduce the computation time taken to shift the map. The three-dimensional nature of the helicopter's environment would still be maintained through the global map. However, removing the third dimension from the local map would greatly reduce the versatility of the local map. If the local map were only a two-dimensional representation, then it would be impossible to gimbal the scanning laser rangefinder, or to improve the sensor to scan in a raster-like pattern. Since both of these options are being considered for future versions of the DSAAV, it may be best to leave the local map as is.

The mapping system currently uses a straightforward 3D adaptation of the Bresenham's line drawing algorithm for tracing the path of the laser in the local map. This algorithm could be modified to draw antialiased lines. Antialiasing involves marking the cells that are close to a line according to their distance from the actual line segment. In computer graphics, this creates a smoother, more natural-looking line. In the local map, this could help to ensure that laser data is represented more accurately in the certainty grid. However, antialiased line drawing takes more time than normal line drawing, and it is debatable whether antialiased line-drawing is more appropriate for certainty grids. Most two-dimensional certainty grid implementations do not use antialiased line-drawing.[15] [5] [16] [8]

One option that was discussed extensively during the design phase of the mapping system was to add some three-dimensional filtering to the local map certainty grid.

Using a discrete, three-dimensional lowpass filter, it would be possible to blur the values of the certainty grid cells so as to more accurately represent the lack of confidence in the position of autonomous helicopter. This idea was originally discarded because of the extra computation required to implement a discrete lowpass filter in three dimensions. However, as processor speeds increase, a blurring process such as this may one day become more feasible.

9.1.2 Global Map Improvements

Currently, the greatest weakness of the global map is its update method. Individually adding cells from the local map to the global map is both time-consuming and inefficient. A better way of updating the global map would be to conglomerate cells from the local map before adding them to the global map. This would speed up the global map update process because larger cells would be inserted into the global map, and would also reduce the amount of conglomeration that needs to be performed after each global map update. The global map addition function has already been written to allow for the addition of larger cells, so all that needs to be done to implement this change is to write a conglomeration function for the local map.

One great thing about the mapping system is that the global map update rate can be set to be any multiple of the local map update rate. This provides a way to relieve computation speed pressures without affecting the local map. One improvement to the global map would be to allow the global map conglomeration rate to be set to be any multiple of the global map update rate. Since conglomerating the global map is such a time-consuming task, this could help to further alleviate computation speed concerns for the global map. On the down side, conglomerating the global map less often would increase the size and reduce the memory efficiency of the global map since like cells would remain unconglomerated for longer periods of time.

Currently, every time the global map is updated, the entire local map is transferred to the global map. One possible improvement to the process of updating the global map would be to copy small portions of the local map to the global map at a faster rate. For example, the first half of the local map would be transferred at t=0, the second half at t=1, then the first half again at t=2, and so on. This way the load of adding and conglomerating cells in the global map would be more well-distributed over time. This could help to reduce the computation load on the autonomous helicopter's on-board processor and allow for more timely updates of the global map.

In the current version of the global map, the occupancy value of a given cell is irrelevant if this cell is a parent. Instead, the occupancy values of the cell's children

are used to determine the occupiedness of the given region. This simplifies the global map update process, but means that the occupancy value for parent cells is completely unused. One possible improvement to the global map would be to have each parent cell store the average occupancy value of its children. This would make it possible for a user of the global map to produce a lower resolution view of a region by only looking at global map cells that are larger than a certain size. This average value could be stored as either an integer or a floating point number, and would most likely be calculated as part of the global map conglomeration function.

As noted earlier, the horizontal, planar nature of the scanning laser rangefinder data reduced the effectiveness of the conglomeration process. I see two ways of improving this. First, the conglomeration could be improved by modifying the kd-tree subdivision and conglomeration algorithms to produce cells that are more flat in the z-dimension, thus more efficiently representing the planar data. Second, the scanning laser rangefinder could be improved so as to scan in both the vertical and horizontal dimensions, possibly by gimballing. This would produce true three-dimensional data for the mapping system to use, which could then more easily be conglomerated by the three-dimensional kd-tree.

The current version of the mapping system only propagates data from the local map the the global map, and not vice versa. Data is not copied from the global map to the local map because of a number of reasons. First, because of the high data rate of the scanning laser rangefinder, there is little need for such propagation; upon moving into a new area the local map is quickly filled with fresh occupancy information. Also, the additional copying process would take up additional computation time, which is at a premium on the DSAAV. Finally, there is a possibility that a careless implementation of such a propagation could create a local map to global map to local map feedback loop that would corrupt cell values. However, as some point in the future it may become advantageous to implement two-way propagation, which would allow the local map to repopulated with information stored in the global map.

Implementing such a transfer from the global map to the local map could also improve the functionality of the DSAAV by making it possible to load a a priori global map (generated from a satellite photo or my other means) which could in turn be propagated into the local map for verification and obstacle avoidance.

If multiple autonomous helicopters are all operating in a given region, it could be advantageous for the helicopters to share a global map. Each autonomous helicopter would still maintain its own local map for integrating and filtering the range measurements from its scanning laser rangefinder, and data from each of these local maps

would be combined in the shared global map. Such a shared global map would allow multiple autonomous helicopters to share obstacle data and coordinate searching a region. Implementing a shared global map would require adding an exclusive lock to the global map addition and conglomeration functions.

9.1.3 Simulation Improvements

As mentioned before, the Draper simulation framework is in a constant state of evolution, so it is likely that there will be many changes to the mapping system and scanning laser rangefinder implementation on the autonomous helicopter.

One simple addition would be to add an option to draw the laser in the helicopter viewing window. This would enable the simulation user to see where the laser rangefinder is scanning and make it easier to debug add-on functions of the mapping system.

The simulation is designed to run a test scenario that includes three autonomous helicopters. The mapping system described in this thesis has been implemented on only the first of these three helicopters. As an extension to the system, the mapping structures could be copied and added to the remaining two autonomous helicopters in the simulation. This addition will be necessary to implement the shared global map discussed in Section 9.1.2.

Soon, the simulation framework will be updated to include full compatibility with the OpenGL language. Once this is done, it should be easier to update the simulation to display local and global map representations that are more visually impressive.

9.2 Uses of the Mapping System

The mapping system is suitable for a variety of uses. A few possible functions of the mapping system are described below. This list is by no means inclusive.

The information stored in the local map should be useful for implementing obstacle avoidance algorithms on the autonomous helicopter. This could be done by reading the local map to determine the locations of solid obstacles and then altering the autonomous helicopter's flight path to avoid these obstacles. The local map is well-suited to obstacle avoidance since it is updated frequently and maintained relative to the autonomous helicopter.

Over time, the autonomous helicopter will have to avoid many obstacles. Using both the local and global maps, it should be possible to implement path planning algorithms for the autonomous helicopter. Such algorithms would obtain obstacle

location data from the mapping system use this to plan safe and efficient flight paths.

If the local map resolution is set to be very precise, then the locations of objects in the local map could be used to augment the autonomous helicopter's estimate of its own position. This is known as landmark-based navigation, and could be done by observing the movements of objects in the local map and extrapolating from this the movement of the autonomous helicopter. In such a system, care must be taken to avoid undesirable feedback between through the navigation position estimate and the shifting of the local map.

9.3 Conclusion

This thesis is one of the first attempts to build a comprehensive mapping system for an autonomous aerial vehicle. Considerable time was spent reviewing mapping systems that have been used previously on autonomous vehicles, and care was taken to ensure that all of Draper's long-term goals were considered in the design of the system. The coordination of the local and global maps represents a novel approach to mapping that may help to improve mapping efficiency and performance. As applied to the DSAAV, this mapping system should provide a solid foundation for future work in obstacle avoidance and path planning.

Appendix A

Glossary

autonomous
According to Webster's Dictionary[2], the word autonomous is defined as "existing or capable of existing independently." The autonomous helicopter maintains flight without human aid, making decisions using an on-board computer and navigating using various sensors.

certainty grid
A type of map in which a region is divided up into equally-sized, uniformly-distributed cells. Each of these cells stores a certainty value. Certainty grids can be used in both two and tree dimensions and are described in Chapter 4, Section 4.1.2.

certainty value
A number indicating the certainty that a cell in a certainty grid is occupied. Typically high certainty values are indicate occupied cells and low certainty values indicate unoccupied cells.

computer graphics
The study displaying images on computer screens.

DSAAV
The Draper Small Autonomous Aerial Vehicle. Otherwise known as the autonomous helicopter.

global map
The large-volume kd-tree tree representation described in detail in Chapter 7. The global map is built from data stored in the local map and is used for mapping the positions of objects over the course of a mission.

kd-tree
A type of hierarchical map similar to a quadtree or octree except that regions are subdivided into two parts instead of four or eight. Kd-trees can be used in either two or three dimensions and are described in Chapter 7.

laser scanner
Another name for a scanning laser rangefinder.

local map
The helicopter-relative certainty grid described in detail in Chapter 6. The local map is used to filter range readings from the scanning laser rangefinder and represent objects that are in the immediate vicinity of the autonomous helicopter.

map
According to Webster's Dictionary[2], a map is defined as

> map- n. [NL mappa, fr. L, napkin, towel] 1a: a representation usu. on a flat surface of the whole or a part of an area 1b: a representation of the celestial sphere or part of it 2: something that represents with a clarity suggestive of a map

This thesis broadens the definition of the word map to include three-dimensional representations of a region of space. Under this expanded definition, some examples of maps are: a 3D certainty grid, an object list for 3D objects, an octree, or a kd-tree.

mapping system
A combination of one or more maps and the functions necessary to make use of these maps. Typical map functions include methods for initializing a map, adding objects to the map, and reading the map.

object list
A type of map in which all the objects in a region are categorized and recorded in a list. Object lists can be used in both two and three dimensions and are described in Chapter 4, Section 4.1.1.

obstacle list
The same thing as an object list.

occupiedness

See "occupancy value."

occupancy value
A numerical measure of the certainty that a region is occupied (as opposed to unoccupied). A high occupancy value indicates that a cell is most likely occupied, whereas a low occupancy value indicates that a call is most likely unoccupied.

octree
A three-dimensional map type which hierarchically divides a volume into smaller and smaller cubes. Each subdivision creates eight smaller cubes. Octrees are described in Chapter 4, Section 4.3.1.

quadtree
A two-dimensional map type which hierarchically divides an area into smaller and smaller squares. Each subdivision creates four smaller squares. Quadtrees are described in Chapter 4, Section 4.3.1.

scanning laser rangefinder
A sensor that uses time of flight measurements of a laser pulse to produce range measurements to objects along the path of the laser. The scanning laser rangefinder built by Long Phan is described in Chapter 2.

Appendix B

PIC Code For Laser Rangefinder

This appendix includes the assembly language code that was used to program the PIC16C73A microcontroller that samples the scanning laser rangefinder and sends data to the test station. The file included below is `laser.asm`; it has been broken into sections for easier reading.

While effort has been made to ensure that this code is free of bugs, no guarantees are made about its correctness. In addition, it is likely that the current version of the `laser.asm` file is different from the one included here.

B.1 PIC Microcontroller Initialization

```
processor 16c73

#include "16c73.h"
#include "laser.h"

;Created 7/22/98 by Rusty Sammon
;Modified 3/22/99 by Rusty Sammon
;This version of the laser.asm code has the following options:
;  * ascii output along serial line
;  * 500Hz laser sampling rate / data send rate
;  * 6.000 Mhz clock speed
;  * configures counter for 500ps resolution
;  * analog servo input for A2D conversion
;  * PWM output to set servo arm position
;
;This PIC code will run a laser rangefinder for use as an altimeter on
;an autonomous helicopter.  The code is reasonably straightforward:
; 1) Start the counter and pulse the laser
; 2) Wait till the counter has data available
; 3) Read the data (parallel data) from the counter
; 4) Send the newest data to computer on a serial line
;
;NOTES
;Timer1 is used to generate an interrupt flag to time the sampling rate
;(500Hz--> 2ms intervals).
;
;Code makes an effort to keep Bank 0 as the active register bank except
;when it is absolutely necessary to use Bank 1.
;
;This version of the code will send the data as Ascii characters,
;which makes it easier to display the data using a Terminal program
;on a PC.
;
;This version of the code contains a startup procedure to configure
;the counter for a 500ps timestep.  The default counter timestep
;is 4ns
```

```
;;;;;;;;;;;;;;;;;
;;; Start Code;;;
;;;;;;;;;;;;;;;;;

ORG 0x0000 ; Reset Start Location
GOTO Startup

ORG 0x0004 ; Interrupt Vector Location
GOTO Interrupt_Handler

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Port Initializations
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Startup:
; Initialize Port A
BCF _rp0
CLRF _porta
BSF _rp0
MOVLW  0xFB
MOVWF  _trisa
; (1 --> input, 0 --> output)
; bit 5 - DATA9 input
; bit 4 - DATA8 input
; bit 3 - not used (analog input- see servo init stuff)
; bit 2 - RCVD output
; bit 1 - not used (analog input- see servo init stuff)
; bit 0 - SERVOIN analog input from servo potentiometer

; Initialize Port B
BCF _rp0
CLRF _portb
BSF _rp0
MOVLW 0xFF
MOVWF _trisb
; Make all of them inputs (1 --> input, 0 --> output)
; data0-data7 are on these pins

; Initialize Port C
BCF _rp0
CLRF _portc
BSF _rp0
MOVLW 0x08
MOVWF _trisc
; bit 7 - STROBE output
; bit 6 - SERIAL output pin
; bit 5 - CLKB output pin
; bit 4 - LASER Output to Laser
; bit 3 - RDY Input from Counter
; bit 2 - DATAIN output to counter
; bit 1 - SERVOUT output to servo
; bit 0 - RESET output to counter

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Initializations for Laser Pulse Rate
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;General Interrupt Setup
BSF _rp0
MOVLW 0xC0
MOVWF _intcon
; bit 7 - Global Interrupts are ENABLED
; bit 6 - Peripheral Interrupts are ENABLED
; bit 5 - Timer 0 (TMR0) overflow interrupt disabled
; bit 4 - RB0 external interrupt disabled
; bit 3 - RB port change interrupt disabled
; bit 2 - TMR0 overflow interrupt flag bit
```

```
; bit 1 - RB0 external interrupt flag bit
; bit 0 - RB port chance interrupt flag bit
;
; Interrupts are used for timing the laser pulse rate
; This program uses the capture compare module flag bit for timing purposes,
; but does not actually perform an interrupt.  The flag bit is set regardless
; of whether interrupts are enabled, and this is all we care about.  The flag
; bit is cleared in the regular code

; Enable Specific Interrupt
BSF _rp0
MOVLW 0x06
MOVWF _pie1
; bit 7 - Parallel Slave port interrupt disabled
; bit 6 - A2D converter interrupt disabled
; bit 5 - USART Receive interrupt disabled
; bit 4 - USART Transmit interrupt disabled
; bit 3 - Synchronous Serial Port interrupt disabled
; bit 2 - Capture Compare module 1 (CCP1) interrupt ENABLED
; bit 1 - TMR2 to PR2 match interrupt ENABLED
; bit 0 - Timer 1 overflow interrupt disabled
;
; Remember, the CCP1 flag bit is used for timing the laser
; pulse rate
; Timer2 is used to time the width of pulses that are sent to the servo
; The frequency of these pulses is controlled by the Pulse_Counter

; Initialize Timer 1
BCF _rp0 ; Use Bank 0
MOVLW 0x01
MOVWF _t1con
; Turn Timer 1 on, it's used for timing the laser pulse rate
; Prescale 1:1
; Use internal clock

; Initialize Capture Compare Module 1 (CCP1)
BCF _rp0
MOVLW 0x0b
MOVWF _ccp1con
; bits 7-6 - Unimplemented
; bits 5-4 - PWM Least Significant bits (not used for ccp1)
; bits 3-0 - Select Compare mode and have interrupt trigger
;            resetting of TMR1
; Changes the interrupt flag when TMR1 has same value as CCPR1H
;   and CCPR1L
; Then resets TMR1 so we can do it over again
; This is used for timing the laser pulse rate

;Load value into CCP1
BCF _rp0
MOVLW 0x0B
MOVWF _ccpr1h
MOVLW 0xB8
MOVWF _ccpr1l
; Get value into compare module for interrupts at correct intervals
; Used for timing the laser pulse rate
; Fosc = 6.000 Mhz
; Clock speed = 1.5Mhz => 667ns (have to recalculate if clock speed is changed)
; 500 samples/sec --> 2ms/sample
; 2ms / 667ns = 3000 = 0x0BB8
; Generates an interrupt every time TMR1 = 0x0BB8

;Reset Interrupt Flags
BCF _rp0
CLRF _pir1 ; Reset Capture Compare module interrupt flag
```

```
BCF Time_For_Loop
; 1 = Interrupt occurred
; 0 = No interrupt

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Initialization for Serial Output
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Initialize Asynchronous Transmitter (USART)
BSF _rp0
MOVLW 0x24
MOVWF _txsta
; bit 7 - Clock Source Select (not used in Asynchronous mode)
; bit 6 - Use 8 bit transmission
; bit 5 - Transmit ENABLED
; bit 4 - Asynchronous mode
; bit 3 - Unimplemented
; bit 2 - High speed transmission
; bit 1 - Transmit Shift Register Status Bit (1=empty, 0=full)
; bit 0 - 9th bit of transmit data (not used)

; Note: Port C, 7 is both the DATAIN output and the Serial port receive input
; Because of this we won't actually enable the serial port (the enable affects
; both the transmit and receive ports) until after loading the control word
; for the counter (this is the only time that DATAIN is used)

; USART Baud Rate Generator Stuff
BSF _rp0
MOVLW 0x09
MOVWF _spbrg
; Value in the SPBRG register controls the baud rate for USART
; transmission using the formula
; Currently, we're using high speed transmission (BRGH=1)
; Baud Rate = Fosc / (16*(X+1))
; Fosc = 6.000 Mhz (have to change this if clock speed changes)
; For baud rate of 37.5kbps, use X=9=0x09

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Initialization for A2D conversion
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Configure analog conversion (ADCON0)
BCF _rp0
MOVLW 0x81
MOVWF _adcon0
; bits 7-6= Conversion clock at Fosc/32= 6.000 Mhz/32= 187.5khz
; This means that it takes 32/4= 8 instructions to do a conversion
; bits 5-3= Analog Channel= 0 (porta,0)
; bit 2 - A/D conversion not yet in progress
; bit 0 - A/D converter module is turned on

; Configure analog port (ADCON1)
BSF _rp0
MOVLW 0x04
MOVWF _adcon1
; bits 7-3= unimplemented
; bits 2-0= set porta bits 0,1,and 3 as analog inputs
; note that only porta,0 is actually used (SERVOIN)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Initialize Servo Controls                                                    '
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Servo_Init:
;set up Timer2 for pulse length timing
BCF _rp0
MOVLW 0x07
MOVWF _t2con
```

```
; bit 7 - unimplemented
; bits 6-3= postscale 1:1
; bit 2 - turn ON timer 2
; bits 1-0= prescale 16

; Interrupts are generated when Timer2==PR2.  We set PR2=the length of the
; servo pulse, and only enable Timer2 once every 50 Hz (the servo update rate)
; Note that we set _pr2=Servo_Destination in laser.h

; Start off with a pulse width equal to Servo_x0
BSF _rp0
MOVLW Servo_x1
MOVWF Servo_Destination
BCF _rp0 ;Switch to Bank 0 (program should always be in bank 0)

;Reset pulse and angle counters
BSF Using_TMR2_Int
CLRF _tmr2
CALL Reset_Servo_Sweep
CALL Start_Servo_Pulse
```

The file `laser.asm` is continued on the next page...

B.2 500ps Counter Initialization

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Load control data into the counter
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Prepare_to_Load_Counter:
BCF RESET ;bring counter reset low to prepare to send reset pulse to counter
BCF LASER ;get laser ready for first pulse
BCF RCVD ;get ready to tell when data is received

Reset_Counter:
BSF RESET ;pulse RESET signal to reset the counter
NOP ;pulse length =  2 cycles = 667ns * 2 = 1.3us
BCF RESET ;finish RESET pulse, counter should be ready to go

Load_Counter_Control_Word:
;The counter has a control data word that needs to be loaded
;at startup to get the counter into the correct mode of operation
;By default, the counter has a timestep of 4ns.  Loading this
;control word reduces the timestep to 500ps.
BCF CLKB ;get set to load the data word
BCF STROBE

;load the first 11 bits of the data word (all zeros)
BCF DATAIN
NOP ;give datain time to be set - necessary?
CALL Pulse_CLKB ;Load data bit 0
CALL Pulse_CLKB ;Load data bit 1
CALL Pulse_CLKB ;Load data bit 2
CALL Pulse_CLKB ;Load data bit 3
CALL Pulse_CLKB ;Load data bit 4
CALL Pulse_CLKB ;Load data bit 5
CALL Pulse_CLKB ;Load data bit 6
CALL Pulse_CLKB ;Load data bit 7
CALL Pulse_CLKB ;Load data bit 8
CALL Pulse_CLKB ;Load data bit 9
CALL Pulse_CLKB ;Load data bit 10

;load the last 5 bits of the data word (all ones)
BSF DATAIN
NOP ;give datain time to be set - necessary?
CALL Pulse_CLKB ;Load data bit 11
CALL Pulse_CLKB ;Load data bit 12
CALL Pulse_CLKB ;Load data bit 13
CALL Pulse_CLKB ;Load data bit 14
CALL Pulse_CLKB ;Load data bit 15

;Finish loading the data word by brining strobe high
BSF STROBE
NOP ;pulse length = 4 cycles = 4 * 667ns = 2.7us
NOP ;cycle 3
NOP ;cycle 4
BCF STROBE

;We're done with DATAIN, now enable serial port (part of RCSTA)
BSF _spen ;Serial port ENABLED
```

The file `laser.asm` *is continued on the next page...*

B.3 The PIC Main Loop

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Main Loop- Do one sample
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Init_Main_Loop:
CLRF _tmr1h ;reset timer #1 for interrupt generation
CLRF _tmr1l ;reset timer #1
BCF Time_For_Loop ;Clear the interrupt flag
BCF _rp0 ;Switch to Bank 0 (program should always be in bank 0)

Main_Loop:
; BSF Start_A2D   ;Sample the servo position and start A2D conversion

; pulse the laser
; clock = 6.000 Mhz = 667ns (need to change code if clock changes)
BSF LASER ;start pulsing the laser
;pulse length = 2 cycle = 2 * 667ns = 1.3us
NOP ;cycle #2
BCF LASER ;finish pulsing the laser

Wait_For_Counter:
BTFSC RDY ;IF counter is ready
GOTO Read_Data ;THEN continue on and read data from counter
BTFSC Time_For_Loop ;ELSE IF interrupt occurs
GOTO Counter_Timeout ;THEN no laser pulse has been received and counter
;has timed out; send blank data and try again
GOTO Wait_For_Counter ;ELSE continue to wait for counter to be ready

Read_Data:
CLRF Data_Low ;Assume all bits are zero until read in
CLRF Data_High
BTFSC data0 ;if bit 0 is set
BSF Data_Low,0 ;then set Data_Low,0
BTFSC data1 ;read bit 1
BSF Data_Low,1
BTFSC data2 ;read bit 2
BSF Data_Low,2
BTFSC data3 ;read bit 3
BSF Data_Low,3
BTFSC data4 ;read bit 4
BSF Data_Low,4
BTFSC data5 ;read bit 5
BSF Data_Low,5
BTFSC data6 ;read bit 6
BSF Data_Low,6
BTFSC data7 ;read bit 7
BSF Data_Low,7
BTFSC data8 ;read bit 8
BSF Data_High,0
BTFSC data9 ;read bit 9
BSF Data_High,1

BSF RCVD ;let counter know that data has been received by pulsing the RCVD output
NOP ;pulse length =  2 cycles = 2 * 667ns = 1.3us
BCF RCVD ;end pulsing RCVD

Send_A2D_Ascii1:
;by now the A2D conversion is easily finished (it takes 8 cycles)
;convert the A2D result to ascii and send it to computer
CLRF Digit
BTFSC _adres, 4
BSF Digit, 0
BTFSC _adres, 5
BSF Digit, 1
```

```
BTFSC _adres, 6
BSF Digit, 2
BTFSC _adres, 7
BSF Digit, 3
CALL Send_Digit_as_Ascii    ;converts digit to ascii and sends it
Send_A2D_Ascii0:
;convert the second digit of the servo position to ascii
CLRF Digit
BTFSC _adres, 0
BSF Digit, 0
BTFSC _adres, 1
BSF Digit, 1
BTFSC _adres, 2
BSF Digit, 2
BTFSC _adres, 3
BSF Digit, 3
CALL Send_Digit_as_Ascii     ;converts digit to ascii and sends it

Send_Space:
MOVLW 0x20 ;Ascii "space"
MOVWF Ascii
CALL Wait_to_Send ;Send the carriage return

Prepare_Ascii2:
;Convert the third digit of the hexadecimal number to ascii
CLRF Digit
BTFSC Data_High, 0
BSF Digit, 0
BTFSC Data_High, 1
BSF Digit, 1
BTFSC Data_High, 2
BSF Digit, 2
BTFSC Data_High, 3
BSF Digit, 3
CALL Send_Digit_as_Ascii ;converts digit to ascii an sends it
Prepare_Ascii1:
;Convert the second digit of the hexadecimal number to ascii
CLRF Digit
BTFSC Data_Low, 4
BSF Digit, 0
BTFSC Data_Low, 5
BSF Digit, 1
BTFSC Data_Low, 6
BSF Digit, 2
BTFSC Data_Low, 7
BSF Digit, 3
CALL Send_Digit_as_Ascii ;converts digit to ascii an sends it
Prepare_Ascii0:
;Convert the first digit of the hexadecimal number to ascii
;This is done by taking the first 4 bits of the datalow register
;and finding their corresponding values as hexadecimal digits
;in ascii
CLRF Digit
BTFSC Data_Low, 0
BSF Digit, 0
BTFSC Data_Low, 1
BSF Digit, 1
BTFSC Data_Low, 2
BSF Digit, 2
BTFSC Data_Low, 3
BSF Digit, 3
CALL Send_Digit_as_Ascii ;converts digit to ascii an sends it

Send_Carriage_Return:
MOVLW 0x0A ;Ascii "line feed"
```

```
MOVWF Ascii
CALL Wait_to_Send ;Send the carriage return

Continue_Servo_Sweep:
DECF Sample_Counter_Low, 1
BTFSC Did_Not_Borrow ;IF there was a borrow
DECF Sample_Counter_High, 1 ;THEN decrement the high byte also

MOVFW Sample_Counter_Low
ADDLW 0x00
BTFSS Result_Zero   ;IF the low byte not zero
GOTO Update_Servo_Position ;THEN Finish the loop
MOVFW Sample_Counter_High ;ELSE test the high byte
ADDLW 0x00
BTFSC Result_Zero ;IF the high byte is zero
CALL Reset_Servo_Sweep ;THEN Start sweeping the other way

Update_Servo_Position:
DECF Pulse_Counter, 1
BTFSC Result_Zero ;IF Pulse_counter == 0
CALL Start_Servo_Pulse ;THEN start the pulse output to servo

Wait_To_Loop:
BTFSS Time_For_Loop ;if interrupt hasn't happened yet
GOTO Wait_To_Loop ;then keep looping
BCF Time_For_Loop ;Clear the interrupt flag
;(Note that this is NOT done in the
;interrupt handler)

Finish_Loop:
CLRWDT ;Clear the Watchdog timer
GOTO Main_Loop ;ELSE continue sweeping this way
```

The file `laser.asm` is continued on the next page...

B.4 PIC Helper Functions

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Counter_Timeout:
; When the counter does not send back a RDY signal before it is time to send
; data, then this procedure is called.  This procedure will clear the interrupt
; and then send a "!" to indicate no data was received.

MOVLW 0x21 ;Send flag value (Ascii "!")
MOVWF Ascii ;to show that no signal is returned
CALL Wait_to_Send ;Send the blank data
GOTO Send_Carriage_Return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Send_Digit_as_Ascii:
; This procedure converts a 4 bit number value in the register Digit
; and converts is to an Ascii value that corresponds to the correct
; hexadecimal digit.  For example,
; (binary 1011)=(hex B)
; converts to
; (Ascii "B")=(hex 42)=(binary 01000010)
;The converted ascii number is stored in the register Ascii

MOVFW Digit ;First, we have to determine if the hexadecimal
SUBLW 0x09 ;digit is a number or a letter.  By subracting
;from 0x09 and then looking at the carry bit, we
;can see if the number is greater than 0x09
;(greater than 0x09 means it's a letter)
BTFSS Overflow ;IF the /borrow bit is 0 (a borrow occurred)
GOTO Convert_Letter ;THEN the digit is a letter
;ELSE the digit is number
Convert_Number:
MOVFW Digit ;Reload the digit, the result of the subraction is useless
ADDLW 0x30  ;0x30 is the ascii value for "0"
;0x31 is the ascii value for "1", etc.
;by adding 0x30, we line up the ascii values for
;digits 0 through 9
MOVWF Ascii
GOTO Wait_to_Send

Convert_Letter:
MOVLW 0x0A ;Subtract 0x0A from the digit to get the offset
SUBWF Digit,0 ;Store this in the working register
ADDLW 0x41  ;0x41 is the ascii value for "A"
;0x42 is the ascii value for "B", etc.
;by adding 0x41, we line up the ascii values for
;letters A through F
MOVWF Ascii

;The Transmit_Flag indicates whether the transmit buffer is empty
;and it is possible to send more data.  It is set regardless of
;whether the transmit interrupt is enabled (in this program, it
;isn't).  The Transmit_Flag can not be cleared in the code.

Wait_to_Send:
NOP

Send_Ascii:
BSF Transmit_Enable ;enable transmission
MOVFW Ascii
MOVWF _txreg
NOP ;wait for it to be loaded into TSR

Wait_to_Finish_Sending:
BTFSS Transmit_Done ;IF Transmit_Flag is NOT set
```

```
GOTO Wait_to_Finish_Sending ;THEN _txreg is not empty and we have to wait
;ELSE _txreg is empty and we can send data
BCF Transmit_Enable ;disable transmission
RETURN

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Pulse_CLKB:
; When loading the data word into the counter, it is necessary to pulse
; the clock for each new bit that is loaded.  This procedure is called
; from the load data word routine to make the above code more readable
; It pulses the clock once and then waits long enough for the clock to
; be ready to be pulsed again.
BSF CLKB ;assume that CLKB is already low for sufficient time
NOP ;pulse length = 2 cycles = 667ns * 2 = 1.3us
NOP
BCF CLKB
NOP ;wait long enough so that Pulse_CLKB can be called again
NOP ;wait length = 2 cycles = 667ns * 2 = 1.3us
RETURN
```

The file `laser.asm` is continued on the next page...

B.5 Interrupt Handling

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Interrupt_Handler:
BCF _rp0
MOVWF W_Save ;Save key registers
MOVFW _status
MOVWF Status_Save
MOVFW _fsr
MOVWF Fsr_Save

BCF _rp0 ;register bank 0
BTFSC Timer2_PR2_Flag    ;IF Timer2 interrupt
CALL Finish_Servo_Pulse ;THEN Handle that
BTFSC CCP1_Flag ;IF CCP1 Interrupt
CALL Handle_CCP1_Int

;use a jump here? so that we don't reset the transmit
;unless absolutely necessary?
BTFSC Transmit_Flag ;IF Transmit int
CALL Handle_Tx_Int
CLRF _txreg ;This is the only way to clear the interrupt flag

BCF A2D_Int_Flag

BCF _rp0
MOVFW Fsr_Save ;Restore key registers
MOVWF _fsr
MOVFW Status_Save
MOVWF _status
MOVFW W_Save
RETFIE ;ELSE do notthing for timer1 interrupts

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Handle_Tx_Int:
MOVLW 0x00
MOVWF _txreg ;clear the flag
BCF Transmit_Enable ;disable transmission
RETURN

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Handle_CCP1_Int:
BCF CCP1_Flag ;Clear the flag
BSF Time_For_Loop ;Mark the variable
RETURN

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Reset_Servo_Sweep:
; The servo destination stores the position that the servo should
; be moving to during the the upcoming sweep.  For now, we only
; use the top 8 bits of the duty cycle register in setting the
; servo position.
BSF _rp0 ;register bank 1
MOVLW Servo_x0 ;Servo_x0 is the start position
SUBWF Servo_Destination, 0
BTFSC Result_Zero ;IF we just went to Servo_x0
GOTO Forward_Sweep ;THEN do a forward sweep
MOVLW Servo_x0 ;ELSE do a reverse sweep
MOVWF Servo_Destination ;move back to Servo_x0
BCF _rp0 ;register bank 0
MOVLW Sweep_Samples_High
MOVWF Sample_Counter_High
MOVLW Sweep_Samples_Low
MOVWF Sample_Counter_Low
RETURN
```

```
Forward_Sweep:
MOVLW Servo_x1
MOVWF Servo_Destination
BCF _rp0 ;register bank 0
MOVLW Sweep_Samples_High
MOVWF Sample_Counter_High
MOVLW Sweep_Samples_Low
MOVWF Sample_Counter_Low
RETURN

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Start_Servo_Pulse:
MOVLW Pulse_Rate ;Reset the pulse_counter
MOVWF Pulse_Counter
CLRF _tmr2 ;reset the timer
BSF SERVOUT ;start pulsing the servo
BCF Timer2_PR2_Flag ;clear the interrupt flag
BSF _rp0
BSF Timer2_Int_Enable ;Turn timer 2 ON
BCF _rp0
BSF Using_TMR2_Int
RETURN

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Finish_Servo_Pulse:
BCF Timer2_PR2_Flag ;clear the interrupt flag
BTFSS Using_TMR2_Int  ;IF we're not using this interrupt
RETURN ;THEN Don't do anything
BCF SERVOUT ;stop pulsing the servo
BSF _rp0
BCF Timer2_Int_Enable ;disable the interrupt
BCF _rp0
BCF Using_TMR2_Int
RETURN

END
```

Appendix C

Mapping Code

This appendix contains all the code for the mapping system functions for both the local map and global map. The mapping functions are defined in the file `mapper.c`, which is included below. All functions are defined in the same file so as to remain consistent with the previously-written Draper simulation code. Section breaks have been added in this appendix to make the the code easier to read.

While effort has been made to ensure that this code is free of bugs, no guarantees are made about its correctness. In addition, it is likely that the current version of the `mapper.c` file is different from the one included here.

This code is included as part of the `aav` version of the Draper simulation. The modified version of the `aav` simulation is stored in the directory `/spirit/disk5/people/rps1681/workarea/aav/`.

C.1 defines, includes, and headers

```
/*************************************************************************
 * mapper.c
 *
 * Mapping software for the scanning laser rangefinder sensor that
 * will be added to the helicopter.
 *
 */

/*constants for angle conversions*/
#define C_DEG2RAD        0.017453292519943295
#define C_DEG2RAD_F      0.017453292519943295f
#define C_RAD2DEG       57.295779513082323
#define C_RAD2DEG_F     57.295779513082323f
#define CUBE_ROOT_ONE_HALF 0.793700526

#include "on_board_ref.h"
/* Contains all the variable definitions from mapper.spech, as well as
 * all the other on_board variable definitions (though this code only
 * uses the nav structure briefly)*/

#include "simio.h"    /*for use of thout during debugging*/
#include <stdlib.h>   /*dynamic memory allocation?*/
#include <math.h>     /*standard math functions*/
#include "mapper.h"
#include "matrix.h"   /*a few basic matrix and vector routines */
#include "sensors_ref.h"  /* the sensor simulation */

/* simple math functions */
#define SQUARE(a) ((a) * (a))
#define ROUND(a) ((((a)-floor(a)) < 0.5) ? (int)floor(a) : (int)ceil(a))
#define MAX(a,b) (((a)>(b))?(a):(b))   /* find maximum of a and b */
#define ABS(a) (((a)<0) ? -(a) : (a))  /* absolute value of a */
#define ZSGN(a) (((a)<0) ? -1 : (a)>0 ? 1 : 0)
```

```
/* line clipping directions */
#define NO_CLIP       0
#define NORTH_SIDE    1
#define SOUTH_SIDE    2
#define EAST_SIDE     3
#define WEST_SIDE     4
#define TOP_SIDE      5
#define BOTTOM_SIDE   6
```

The file `mapper.c` *is continued on the next page...*

C.2 Local Map Helper Functions

```
/*************************************************************************/
local_map_cell *get_local_map_cell (struct local_map_ref *map,
    int x, int y, int z)
/* don't bother with error checking- it just slows things down,
 * and returning a NULL will only cause an error later in the program */
{
  return ( &(map->cells [x*map->y_cells*map->z_cells + y*map->z_cells + z]) );
}

/*************************************************************************/
int get_local_map_cell_index (int x_cells, int y_cells, int z_cells,
      int x, int y, int z)
{
  return (x*y_cells*z_cells + y*z_cells + z);
}

/*************************************************************************/
void print_local_map (struct local_map_ref *map)
/* for debugging purposes*/
{
  int i,j,k;
  thout("******* Printing the local map: *******\n");
  for (j=0; j<map->y_cells; j++) {
    thout ("level y=%d\n", j);
    for (k=0; k<map->z_cells; k++) {
      for (i=0; i<map->x_cells; i++) {
thout("%4d", get_local_map_cell(map, i,j,k)->occupied);
      }
      thout("\n");
    }
  }
  thout("******* done printing the local map: *******\n");
  exec_pause("");
}
```

The file mapper.c is continued on the next page...

C.3 Local Map Initialization

```
/*********************************************************************/
void init_local_map (struct mapperIn_ref *in,
     struct local_map_ref *map)
     /*allocate memory and make the parameter*/
{
  /*memory management, if necessary*/
  if (map->reinitialize) {
    free (map->cells);
    map->reinitialize= OFF;
    map->reinit_buffer= ON;
  }

  /*allocating the memory for the map*/
  map->cells= calloc (map->x_cells * map->y_cells * map->z_cells,
      sizeof(struct local_map_cell));

  /*compute the initial helicopter offset in feet*/
  map->x_offset= map->x_desired_offset;
  map->y_offset= map->y_desired_offset;
  map->z_offset= map->z_desired_offset;

  /*setting the map location and orientation*/
  map->x= in->heli_x - map->x_offset + in->sensor_x;
  map->y= in->heli_y - map->y_offset + in->sensor_y;
  map->z= in->heli_z - map->z_offset + in->sensor_z;
}
```

The file `mapper.c` *is continued on the next page...*

C.4 Fetching Local Map Inputs

```
/*********************************************************************/
void get_mapper_inputs (struct mapperIn_ref *in,
struct aavnavigation_ref *nav)
/* Fetches the map input data, which is the range data that is sent along
 * a serial line from the scanning laser rangefinder.
 *
 * REAL VERSION
 * The scanning laser rangefinder (built by Long Phan) produces at stream
 * of range data that is transmitted to the helicopter along a serial line.
 * Data rate, packet size, and format are currently undetermined.  This
 * procedure will read the data from the scanning laser rangefinder and
 * store it into the input array.
 *
 * SIM VERSION
 * I'll figure out some way to create phony data
 *
 * NOTE: I'll want to add in the offset and orientation of the sensor
 * on the helicopter.
 */

{
  struct aavsensors_laser_ref *laser= &aavsensors_laser;
  int i;  /*the current input range from the hardware*/

  /*get vehicle position and orientation estimates from nav filter*/
  in->heli_x= nav->pos[0];
  in->heli_y= nav->pos[1];
  in->heli_z= nav->pos[2];
  in->heli_phi= nav->phi;
  in->heli_theta= nav->theta;
  in->heli_psi= nav->psi;

#ifdef SIM_VERSION
  for (i=0; i<SWEEP_SAMPLES; i++) {
    in->angle_counts[i]= laser->angle_count[i];
    in->range_counts[i]= laser->range_count[i];
  }
#else /* real version */
  /*get range data from the hardware (sent as a count number)*/
  /*want to add in start and stop bytes, etc*/
  for (i=0; i<SWEEP_SAMPLES; i++) {
    /*
     *
     * Insert code here
     *
     */
  }
#endif

  /*convert the count values to ranges and angles*/
  for (i=0; i<SWEEP_SAMPLES; i++) {
    in->angles[i]= in->angle0 + in->angle_counts[i] * in->angle_res;
    in->ranges[i]= in->minRange + in->range_counts[i] * in->range_res;
  }
}
```

The file `mapper.c` *is continued on the next page...*

C.5 Shifting the Local Map

```
/*********************************************************************/
void move_local_map (struct mapperIn_ref *in,
     struct local_map_ref *map,
     double heli_orient[3][3])
/* Translate the local map to the new helicopter position.
 * This maintains the map in the same frame of reference as the helicopter
 *
 * 1) convert cell coordinates--  (0,0,0) is above, behind and left of
 * heli -- to heli coordinates (everything relative to heli).
 * NOTE that while cell coords have changed, no cells have been shifted
 * 3) translate according to the heli shift
 *
 * TRANSLATION
 * The local map only needs to be translated if the helicopter has moved
 * outside of it's original cell in the local map.  Otherwise, we can
 * keep the local map in the same place (no translation), since this is
 * a good aproximation to the helicopter positon.  Note that this assumes
 * that the cell size is not much greater than the heli size.
 *
 */
  /*everything is done in units of cells rather than feet*/
{
  /*constants*/
  int x_cells= map->x_cells;
  int y_cells= map->y_cells;
  int z_cells= map->z_cells;
  float cell_size= map->cell_size;

  /*variables*/
  double sensor_heli_offset[3];    /*offset of the sensor in heli frame*/
  double sensor_global_offset[3];  /*sensor offset from heli in global frame*/
  double newX, newY, newZ;                  /* new sensor position, in feet*/
  double desiredX, desiredY, desiredZ;      /* desired sensor position,in feet*/
  double deltaX, deltaY, deltaZ;            /* sensor shift distance, in feet */
  int cell_deltaX, cell_deltaY, cell_deltaZ; /* map shift distance, in cells */
  double map_shiftX, map_shiftY, map_shiftZ; /* map shitt distance, in feet */

  /*shifting*/
  int i,j,k;                    /* origin for cell being moved*/
  int x,y,z;                    /* destination for cell being moved*/
  local_map_cell *temp_cells;
                 /* temp copy of map cells used for shifting */

  /* need to incorporate sensor x, y, z relative to helicopter orientation */
  sensor_heli_offset[0]= in->sensor_x;
  sensor_heli_offset[1]= in->sensor_y;
  sensor_heli_offset[2]= in->sensor_z;
  matrix_times_vector (heli_orient, sensor_heli_offset, sensor_global_offset);

  /* find the new sensor position in global frame*/
  newX= in->heli_x + sensor_global_offset[0];
  newY= in->heli_y + sensor_global_offset[1];
  newZ= in->heli_z + sensor_global_offset[2];

  /* find the desired sensor position in global frame*/
  desiredX= map->x + map->x_desired_offset;
  desiredY= map->y + map->y_desired_offset;
  desiredZ= map->z + map->z_desired_offset;

  /* determine the deviation in the sensor position */
  deltaX= newX - desiredX;
  deltaY= newY - desiredY;
  deltaZ= newZ - desiredZ;
```

```
  /* if we haven't deviated more than max_deviation, save the new heli
   * position and return */
  if ((sqrt(SQUARE(deltaX) + SQUARE(deltaY) + SQUARE(deltaZ)))
      < ((double)map->max_deviation)) {
    map->x_offset = newX - map->x;
    map->y_offset = newY - map->y;
    map->z_offset = newZ - map->z;
    return;
  }

  /* compute the amount to shift the map */
  cell_deltaX= (int) ROUND (deltaX / cell_size);
  cell_deltaY= (int) ROUND (deltaY / cell_size);
  cell_deltaZ= (int) ROUND (deltaZ / cell_size);
  map_shiftX= cell_deltaX * cell_size;
  map_shiftY= cell_deltaY * cell_size;
  map_shiftZ= cell_deltaZ * cell_size;

  /*mark the change in position of the local map*/
  map->x += map_shiftX;
  map->y += map_shiftY;
  map->z += map_shiftZ;
  map->x_offset = newX - map->x;
  map->y_offset = newY - map->y;
  map->z_offset = newZ - map->z;

  /*allocate memory for temporary map cells*/
  temp_cells= calloc (x_cells * y_cells * z_cells,
      sizeof(struct local_map_cell));

  /* SHIFT THE CELLS- cell (i,j,k) shifts to cell (x,y,z)*/
  x= -cell_deltaX;
  for (i=0; i<x_cells; i++) {
    y= -cell_deltaY;
    for (j=0; j<y_cells; j++) {
      z= -cell_deltaZ;
      for (k=0; k<z_cells; k++) {
  if ((x >= 0) && (x < x_cells) &&
      (y >= 0) && (y < y_cells) &&
      (z >= 0) && (z < z_cells)) {
  COPY_CELL (
    get_local_map_cell (map, i, j, k),
    &temp_cells [get_local_map_cell_index (x_cells, y_cells, z_cells,
   x, y, z)]);
}
z++;
      }
      y++;
    }
    x++;
  }

  /* copy the tempmap back into the real local map*/
  for (i=0; i<x_cells; i++)
    for (j=0; j<y_cells; j++)
      for (k=0; k<z_cells; k++) {
COPY_CELL (
  &temp_cells [get_local_map_cell_index (x_cells, y_cells, z_cells,
 i, j, k)],
  get_local_map_cell (map, i, j, k));
      }
}
```

The file `mapper.c` *is continued on the next page...*

C.6 Local Map Line Clipping

```
/*********************************************************************/
void map_clip_line3d (struct local_map_ref *map,
      double *x_orig, double *y_orig, double *z_orig,
      double beam_vector[3],
      int *clip)
/* This algorithm assumes that the origin of the line (where the
 * helicopter is) is within the map.  The algorithm then clips the end
 * point of the line using a modified version of the Cohen-Sutherland
 * Line-Clipping algorithm
 */
{
  double x=*x_orig;
  double y=*y_orig;
  double z=*z_orig;

  /* clip along x dimension */
  if (x < 0) {
    y+= (0 - x) * beam_vector[1]/beam_vector[0];
    z+= (0 - x) * beam_vector[2]/beam_vector[0];
    x= 0;
    *clip=1;
  }
  else if (x >= map->x_cells) {
    y+= (map->x_cells - x) * beam_vector[1]/beam_vector[0];
    z+= (map->x_cells - x) * beam_vector[2]/beam_vector[0];
    x= map->x_cells -1;
    *clip=1;
  }
  /* clip along y dimension */
  if (y < 0) {
    x+= (0 - y) * beam_vector[0]/beam_vector[1];
    z+= (0 - y) * beam_vector[2]/beam_vector[1];
    y= 0;
    *clip=1;
  }
  else if (y >= map->y_cells) {
    x+= (map->y_cells - y) * beam_vector[0]/beam_vector[1];
    z+= (map->y_cells - y) * beam_vector[2]/beam_vector[1];
    y= map->y_cells -1;
    *clip=1;
  }
  /* clip along z dimension */
  if (z < 0) {
    x+= (0 - z) * beam_vector[0]/beam_vector[2];
    y+= (0 - z) * beam_vector[1]/beam_vector[2];
    z= 0;
    *clip=1;
  }
  else if (z >= map->z_cells) {
    x+= (map->z_cells - z) * beam_vector[0]/beam_vector[2];
    y+= (map->z_cells - z) * beam_vector[1]/beam_vector[2];
    z= map->z_cells -1;
    *clip=1;
  }

  *x_orig= x;
  *y_orig= y;
  *z_orig= z;
}
```

The file `mapper.c` *is continued on the next page...*

C.7 Local Map Cell Update Rules

```
/*********************************************************************/
void mark_open_cell (struct local_map_ref *map,
    int x, int y, int z)
{
  local_map_cell *cell= get_local_map_cell(map,x,y,z);

  cell->occupied= cell->occupied + map->miss_points;
}

/*********************************************************************/
void mark_hit_cell (struct local_map_ref *map,
    int x, int y, int z)
{
  local_map_cell *cell= get_local_map_cell(map,x,y,z);

  cell->occupied= cell->occupied + map->hit_points;
}
```

The file `mapper.c` *is continued on the next page...*

C.8 Local Map Line Drawing

```
/*****************************************************************/
void map_mark_line3d (struct local_map_ref *map,
      int x1, int y1, int z1, int x2, int y2, int z2)
/* line3d was dervied from DigitalLine.c published as "Digital Line Drawing"
 * by Paul Heckbert from "Graphics Gems", Academic Press, 1990
 *
 * 3D modifications by Bob Pendleton. The original source code was in the
 * public domain, the author of the 3D version places his modifications in the
 * public domain as well.
 *
 * line3d uses Bresenham's algorithm to generate the 3 dimensional points on a
 * line from (x1, y1, z1) to (x2, y2, z2)
 */
{
    int xd, yd, zd;  /*decision vars for each direction*/
    int x, y, z;
    int ax, ay, az;
    int sx, sy, sz;
    int dx, dy, dz;

    dx = x2 - x1;      /* deltas */
    dy = y2 - y1;
    dz = z2 - z1;
    ax = ABS(dx) << 1; /* 2 times abs(dx)*/
    ay = ABS(dy) << 1;
    az = ABS(dz) << 1;
    sx = ZSGN(dx);     /* the sign of dx */
    sy = ZSGN(dy);
    sz = ZSGN(dz);
    x = x1;            /* the start point for the line */
    y = y1;
    z = z1;

    if (ax >= MAX(ay, az)) {          /* x dominant */
      yd = ay - (ax >> 1);
      zd = az - (ax >> 1);

      for (;;) {
if (x == x2) {  /*don't mark the last point in the line*/
  return;
}
/* mark points along the line as misses */
mark_open_cell (map,x,y,z);
if (yd >= 0) {
  y += sy;
  yd -= ax;
}
if (zd >= 0) {
  z += sz;
  zd -= ax;
}
x += sx;
yd += ay;
zd += az;
      }
    }
    else if (ay >= MAX(ax, az)) {          /* y dominant */
      xd = ax - (ay >> 1);
      zd = az - (ay >> 1);
      for (;;) {
if (y == y2) { /*don't mark the last point in the line*/
  return;
}
```

```
/* mark points along the line as misses */
mark_open_cell (map,x,y,z);
if (xd >= 0) {
  x += sx;
  xd -= ay;
}
if (zd >= 0) {
  z += sz;
  zd -= ay;
}
y += sy;
xd += ax;
zd += az;
      }
    }
    else if (az >= MAX(ax, ay)) {           /* z dominant */
      xd = ax - (az >> 1);
      yd = ay - (az >> 1);
      for (;;) {
if (z == z2) { /*don't mark the last point in the line*/
  return;
}
/* mark points along the line as misses */
mark_open_cell (map,x,y,z);
if (xd >= 0) {
  x += sx;
  xd -= az;
}
if (yd >= 0) {
  y += sy;
  yd -= az;
}
z += sz;
xd += ax;
yd += ay;
      }
    }
}
```

The file `mapper.c` *is continued on the next page...*

C.9 Local Map Update Function

```
/*************************************************************************/
void update_local_map (struct mapperIn_ref *in,
      struct local_map_ref *map)
/* 1) translate the local map to be at the helicopter's new position.
 *
 * 2) degrade the local map according to the uncertainty in the helicopter's
 * position and orientation.
 *
 * 3) put the new range data from the scanning laser rangefinder into the
 * local map.
 *
 * NOTE: May want to move the map less often and therefore have the
 * heli_offset in the map vary a little bit.  This would be cool because
 * it would mean that we don't have to move the map every time.
 *
 * NOTE: will want to save the cosines once I figure out if these are
 * the right ones.
 */
{
  int i;        /*the current sample*/
  int clip;     /*did we have to clip the line to fit it in the map?*/
  float range;  /*current range that is being added to map*/
  float cell_size= map->cell_size;
  int heliX= ROUND((map->x_offset-0.5)/cell_size); /*heli cell in local map*/
  int heliY= ROUND((map->y_offset-0.5)/cell_size); /*heli cell in local map*/
  int heliZ= ROUND((map->z_offset-0.5)/cell_size); /*heli cell in local map*/
  double x, y, z;  /*target position in local map*/
  int x_int, y_int, z_int;  /*target position in local map, rounded*/

  /* rotation matrices */
  double heli_orient[3][3];   /*helicopter in free space*/
  double sensor_orient[3][3]; /*sensor on the helicopter*/
  double beam_orient[3][3];   /*beam coming out of the sensor*/
  /* unit vectors */
  double orig_vector[3] = {1.0, 0.0, 0.0};  /*x axis is zero yaw,pitch,roll*/
  double heli_vector[3];      /*unit vector in heli direction*/
  double sensor_vector[3];    /*unit vector in sensor direction*/
  double beam_vector[3];      /*unit vector in beam direction*/


  /*reinialize the map if size has been changed by user*/
  if (map->reinitialize == ON)  init_local_map (in, map);

  /*calculate orientation of 1)helicopter and 2) sensor on the heli*/
  set_rotation_matrix(heli_orient, in->heli_phi, in->heli_theta, in->heli_psi);
  matrix_times_vector(heli_orient, orig_vector, heli_vector);
  set_rotation_matrix(sensor_orient, in->sensor_phi, in->sensor_theta,
      in->sensor_psi);
  matrix_times_vector(sensor_orient, heli_vector, sensor_vector);

  /*move the local map to it's new position*/
  move_local_map (in, map, heli_orient);

  /*put the new range data into the local map*/
  for (i=0; i<SWEEP_SAMPLES; i++) {
    /* sensor scans in yaw only*/
    set_rotation_matrix(beam_orient, 0.0f, 0.0f, in->angles[i]);
    matrix_times_vector(beam_orient, sensor_vector, beam_vector);

    /* Clip the range line so that it lies entirely within the local map */
    range= in->ranges[i];
    clip=0;
    if (range >= in->maxRange) {
```

```
      clip=1;
      range= in->maxRange;
    }

    /* compute the final point of the line using the actual heli position*/
    x= (map->x_offset + range*beam_vector[0]) / cell_size;
    y= (map->y_offset + range*beam_vector[1]) / cell_size;
    z= (map->z_offset + range*beam_vector[2]) / cell_size;

    /* clip the final point of the line so that it lies within the local map */
    map_clip_line3d (map, &x, &y, &z, beam_vector, &clip);

    /* round to the nearest integer for line marking purposes */
    x_int= (int)ROUND(x -0.5);
    y_int= (int)ROUND(y -0.5);
    z_int= (int)ROUND(z -0.5);

   /* mark open cells along the path to the hit object */
    map_mark_line3d (map, heliX, heliY, heliZ, x_int, y_int, z_int);
    if (!clip)
      mark_hit_cell (map,x_int,y_int,z_int); /*mark the final cell as hit*/
    else
      mark_open_cell (map,x_int,y_int,z_int); /*mark the final cell as miss*/
  }
}
```

The file `mapper.c` *is continued on the next page...*

C.10 Global Map Helper Functions

```
/*************************************************************************/
void free_global_map_cells (global_map_cell *tree)
/* reallocates memory used by the children of a global map cell*/
{
  if (tree->split != NONE) {
    free_global_map_cells (tree->child0);
    free(tree->child0);
    free_global_map_cells (tree->child1);
    free(tree->child1);
  }
}

/*************************************************************************/
void print_tree_cell (global_map_cell *tree)
{
  thout ("_______ GLOBAL MAP TREE CELL %x _____________________\n", tree);
  if (tree == NULL) {
    thout ("empty\n");
    thflush();
    return;
  }
  thout ("extends from (%.1f,%.1f,%.1f) to (%.1f,%.1f,%.1f)\n",
 tree->x0,tree->y0,tree->z0, tree->x1,tree->y1,tree->z1);
  thout ("occupied= %d    ",tree->occupied);
  switch (tree->split) {
  case NONE: thout ("split= NONE\n"); break;
  case X: thout ("split= X\n"); break;
  case Y: thout ("split= Y\n"); break;
  case Z: thout ("split= Z\n"); break;
  }
  thout ("parent=%x, child0=%x, child1=%x\n",
 tree->parent, tree->child0, tree->child1);
  thout("\n");
  thflush();
}

/*************************************************************************/
void print_global_map_tree (global_map_cell *tree)
{
  if (tree != NULL) {
    print_tree_cell (tree);
    if (tree->split != NONE) {
      print_global_map_tree (tree->child0);
      print_global_map_tree (tree->child1);
    }
  }
}
```

The file `mapper.c` *is continued on the next page...*

C.11 Global Map Tree Manipulation

```
/*****************************************************************************/
void global_map_delete_tree (global_map_cell *tree)
/* RECURSIVE */
{
  if (tree == NULL) return;
  if (tree->split != NONE) {  /*there is a split in the tree*/
    global_map_delete_tree(tree->child0);
    global_map_delete_tree(tree->child1);
  }
  free(tree);
}

/*****************************************************************************/
void global_map_copy_tree (global_map_cell *orig,
   global_map_cell **copy)
/* RECURSIVE
 * the global map is copied, cell-by-cell, into a entirely
 * new global map.  Note that the parent pointer for each tree
 * cell is assigned by the parent itself (rephrased: cell A
 * sets the parent pointers in each of its children).  The
 * base of the tree marks its own parent as NULL.
 */
{
  if (orig == NULL)    *copy= NULL;
  else {
    *copy= malloc(sizeof(global_map_cell));
    (*copy)->occupied= orig->occupied;
    (*copy)->x0= orig->x0;
    (*copy)->y0= orig->y0;
    (*copy)->z0= orig->z0;
    (*copy)->x1= orig->x1;
    (*copy)->y1= orig->y1;
    (*copy)->z1= orig->z1;
    (*copy)->split= orig->split;

    if (orig->split == NONE) {
      (*copy)->child0= NULL;
      (*copy)->child1= NULL;
    }
    else { /*there is a split in the tree*/
      global_map_copy_tree (orig->child0, &((*copy)->child0));
      global_map_copy_tree (orig->child1, &((*copy)->child1));
      (*copy)->child0->parent= (*copy);
      (*copy)->child1->parent= (*copy);
    }

    /* base case for a the base cell (which has no parent) */
    if (orig->parent == NULL)
      (*copy)->parent= NULL;
  }
}
```

The file mapper.c is continued on the next page...

C.12 Global Map Initialization

```
/*********************************************************************/
void init_global_map (struct global_map_ref *map)
{
  /*memory management, if necessary*/
  if (map->reinitialize) {
    free_global_map_cells (map->tree);
    map->total_cells= 0;
    map->reinitialize= OFF;
  }

  /*initialize the head of the global map kd tree*/
  map->tree= malloc (sizeof(global_map_cell));
  map->tree->occupied= map->init_occupied;
  map->tree->x0= map->init_x0;
  map->tree->y0= map->init_y0;
  map->tree->z0= map->init_z0;
  map->tree->x1= map->init_x1;
  map->tree->y1= map->init_y1;
  map->tree->z1= map->init_z1;
  map->tree->split= NONE;
  map->tree->parent= NULL;
  map->tree->child0= NULL;
  map->tree->child1= NULL;
}
```

The file mapper.c is continued on the next page...

C.13 Global Map Cell Splitting

```
/*********************************************************************/
enum dimension get_next_gmap_split (struct global_map_ref *map,
    global_map_cell *tree)
/* tell the tree to split along it's greatest dimension.  If the tree
   is too small to be split, then return NONE */
{
  float double_min_size= 2.0* map->min_cell_size;
  float x_span= tree->x1 - tree->x0;
  float y_span= tree->y1 - tree->y0;
  float z_span= tree->z1 - tree->z0;
  enum dimension split;  /*the answer*/

  if (x_span > y_span)
    if (x_span > z_span)
      if (x_span >= double_min_size)
split= X;
      else
split= NONE;
    else
      if (z_span >= double_min_size)
split= Z;
      else
split= NONE;
  else
    if (y_span > z_span)
      if (y_span >= double_min_size)
split= Y;
      else
split= NONE;
    else
      if (z_span >= double_min_size)
split= Z;
      else
split= NONE;
  return (split);
}

/*********************************************************************/
int gmap_add_children (struct global_map_ref *map,
       global_map_cell *tree)
/* both children should be uninitialized
 * RETURNS an int value indicating whether it was possible to add children
 * to the tree 1=success 0=failure */
{
  global_map_cell *child0, *child1;
  float split_plane;  /*coordinate of the split*/

  /*first, make sure that the tree cell is large enough to split*/
  tree->split= get_next_gmap_split (map, tree);
  if (tree->split == NONE)
    return(0);

  child0= tree->child0= malloc(sizeof(global_map_cell));
  child1= tree->child1= malloc(sizeof(global_map_cell));
  child0->x0= tree->x0;    child0->y0= tree->y0;    child0->z0= tree->z0;
  child0->x1= tree->x1;    child0->y1= tree->y1;    child0->z1= tree->z1;
  child1->x0= tree->x0;    child1->y0= tree->y0;    child1->z0= tree->z0;
  child1->x1= tree->x1;    child1->y1= tree->y1;    child1->z1= tree->z1;

  if (tree->split == X) {
    split_plane= (tree->x0 + tree->x1) / 2.0f;
    child0->x1= split_plane;
    child1->x0= split_plane;
```

```
  } else if (tree->split == Y) {
    split_plane= (tree->y0 + tree->y1) / 2.0f;
    child0->y1= split_plane;
    child1->y0= split_plane;
  } else if (tree->split == Z) {
    split_plane= (tree->z0 + tree->z1) / 2.0f;
    child0->z1= split_plane;
    child1->z0= split_plane;
  }
  child0->parent=    child1->parent=    tree;
  child0->split=     child1->split=     NONE;
  child0->child0=    child1->child0=    NULL;
  child0->child1=    child1->child1=    NULL;
  child0->occupied=  child1->occupied=  map->init_occupied;

  map->total_cells++;  /*helps with debugging*/
  return(1);
}
```

The file **mapper.c** *is continued on the next page...*

C.14 Global Map Cell Addition

```
/*********************************************************************/
void global_map_mark_cell (struct global_map_ref *map,
   global_map_cell *tree,
   int occupied)
/* RECURSIVE
 * This function is called when the cell sizes are the same.
 * But this doesn't necessarily mean that the tree doesn't have children.
 * Mark the tree and all of it's decendents according to the
 * occupied value.
 */
{
  tree->occupied+= occupied;
  if (tree->split != NONE) {
    global_map_mark_cell (map, tree->child0, occupied);
    global_map_mark_cell (map, tree->child1, occupied);
  }
}

/*********************************************************************/
void global_map_add_cell (struct global_map_ref *map,
  global_map_cell *tree,
  float x0, float y0, float z0,
  float x1, float y1, float z1,
  int occupied)
/* RECURSIVE
 * Possible cases:
 *    1) New cell is much bigger than current tree cell.  We assume
 *       away this case (it will have already been handled higher
 *       up in the tree) and treat it like case 2.
 *    2) New cell is too small (shouldn't happen).  Ignore it.
 *    3) New cell is the same size (within one min_cell_size) as the tree
 *       cell.  Mark the current cell accordingly.
 *    4) New cell is smaller than the tree cell (but not too small).  Split
 *       the tree cell (if this hasn't been done already), then split the
 *       new cell along the same boundary.  Recurse on each of the two
 *       children.
 *
 * An input cell will be considered to fill a minimum level tree cell if
 * the input cell's volume is great than one half of the volume of the
 * tree cell.  If everything is cubes (which may not be the case), then
 * this test is equivalent to testing if a the input cell's edge length
 * is greater than the cube root of (0.5*min_cell_size).  For non-cubes,
 * we test that the minimum edge length of the input cell is greater
 * than this quantity.  We test this at each split, guaranteeing that we
 * won't map out an entire tree just to find that the cell has been
 * reduced to such a small size that it is insignificant.
 */
{
  float min_cell_size= map->min_cell_size;
  float roundup= CUBE_ROOT_ONE_HALF * min_cell_size;
  float rounddown= min_cell_size - roundup;
  float split_plane; /*the coordinate at which the cell is split */

#if 0
  thout ("\nlocal cell (%.1f,%.1f,%.1f)->(%.1f,%.1f,%.1f)\n",
 x0,y0,z0, x1,y1,z1);
  thout ("tree cell (%.1f,%.1f,%.1f)->(%.1f,%.1f,%.1f) address=%x\n",
 tree->x0,tree->y0,tree->z0, tree->x1,tree->y1,tree->z1,tree);
#endif

  /* 2) For security sake, stop if the size of the current tree cell
   * is less than that minimum cell size.  This should never happen,
   * if the rest of the code works correctly.
```

```
 */
if (((tree->x1 - tree->x0) < min_cell_size) ||
    ((tree->y1 - tree->y0) < min_cell_size) ||
    ((tree->z1 - tree->z0) < min_cell_size)) {
  thout ("Error: tree cell too small; recursed too far\n");
  thflush();
  return;
}

/* 3) mark the current cell  and stop if: all cell vertices are outside
 * treecell vertices or within rounddown of the treecell verices   */
if ((x0 < (tree->x0 + rounddown)) &&
    (y0 < (tree->y0 + rounddown)) &&
    (z0 < (tree->z0 + rounddown)) &&
    (x1 >= (tree->x1 - rounddown)) &&
    (x1 >= (tree->x1 - rounddown)) &&
    (x1 >= (tree->x1 - rounddown))) {
  global_map_mark_cell (map, tree, occupied);
  return;
}

/* 4) split the cell.  Each child from the split should only be used
 * so long as  min edge of child > roundup   */
if (tree->split == NONE) {
  if (!gmap_add_children (map, tree)) {
    /*if tree cell is too small to have children, just mark it anyway*/
    global_map_mark_cell (map, tree, occupied);
    return;
  }
}

if (tree->split == X) { /*************** SPLIT ALONG X DIMENSION **********/
  split_plane= (tree->x0 + tree->x1) / 2.0f;

  /* first, check if the new cell even needs to be broken up */
  if ((x0 < split_plane + rounddown) && (x1 < split_plane + rounddown)) {
    global_map_add_cell (map, tree->child0, x0,y0,z0, x1,y1,z1, occupied);
    return;
  }
  else
  if ((x0 >= split_plane - rounddown) && (x1 >= split_plane - rounddown)) {
    global_map_add_cell (map, tree->child1, x0,y0,z0, x1,y1,z1, occupied);
    return;
  }
  /* Now we know that the new cell must be broken into parts, and that
   * these parts are both big enough.*/
  global_map_add_cell (map, tree->child0, x0,y0,z0,
split_plane,y1,z1, occupied);
  global_map_add_cell (map, tree->child1, split_plane,y0,z0,
x1,y1,z1, occupied);
}

else if (tree->split == Y) { /********** SPLIT ALONG Y DIMENSION **********/
  split_plane= (tree->y0 + tree->y1) / 2.0f;

  /* first, check if the new cell even needs to be broken up */
  if ((y0 < split_plane + rounddown) && (y1 < split_plane + rounddown)) {
    global_map_add_cell (map, tree->child0, x0,y0,z0, x1,y1,z1, occupied);
    return;
  }
  else
  if ((y0 >= split_plane - rounddown) && (y1 >= split_plane - rounddown)) {
    global_map_add_cell (map, tree->child1, x0,y0,z0, x1,y1,z1, occupied);
    return;
  }
```

```
    /* Now we know that the new cell must be broken into parts, and that
     * these parts are both big enough.*/
    global_map_add_cell (map, tree->child0, x0,y0,z0,
 x1,split_plane,z1, occupied);
    global_map_add_cell (map, tree->child1, x0,split_plane,z0,
 x1,y1,z1, occupied);
  }

  else if (tree->split == Z) { /********** SPLIT ALONG Z DIMENSION **********/
    split_plane= (tree->z0 + tree->z1) / 2.0f;

    /* first, check if the new cell even needs to be broken up */
    if ((z0 < split_plane + rounddown) && (z1 < split_plane + rounddown)) {
      global_map_add_cell (map, tree->child0, x0,y0,z0, x1,y1,z1, occupied);
      return;
    }
    else
    if ((z0 >= split_plane - rounddown) && (z1 >= split_plane - rounddown)) {
      global_map_add_cell (map, tree->child1, x0,y0,z0, x1,y1,z1, occupied);
      return;
    }
    /* Now we know that the new cell must be broken into parts, and that
     * these parts are both big enough.*/
    global_map_add_cell (map, tree->child0, x0,y0,z0,
 x1,y1,split_plane, occupied);
    global_map_add_cell (map, tree->child1, x0,y0,split_plane,
 x1,y1,z1, occupied);
  }
}
```

The file `mapper.c` *is continued on the next page...*

C.15 Global Map Conglomeration

```
/*********************************************************************/
void conglomerate_tree (struct global_map_ref *map,
global_map_cell *parent)
{
  global_map_cell *child0, *child1;

  /*base case- there's nothing to conglomerate here*/
  if (parent->split == NONE)
    return;

  child0= parent->child0;
  child1= parent->child1;
  conglomerate_tree(map, child0);
  conglomerate_tree(map, child1);

  /* you can only conglomerate children, not grandchildren*/
  if ((child0->split != NONE) ||
      (child1->split != NONE))
    return;

#if 0
  int occupied_merge  na     10 :merge threshold for occupied cells;
  int unoccupied_merge  na  -10 :merge threshold for occupied cells;
      ((child0->occupied > map->occupied_merge) &&
       (child1->occupied > map->occupied_merge)) ||
      ((child0->occupied < map->unoccupied_merge) &&
       (child1->occupied < map->unoccupied_merge))) {

#endif

  /* children need to have the same occupiedness */
  if (ABS(child0->occupied - child1->occupied) <= map->merge_tolerance) {
    parent->occupied= (int) (child0->occupied + child1->occupied) / 2;
    free (child0);
    free (child1);
    parent->child0= NULL;
    parent->child1= NULL;
    parent->split=NONE;
    map->total_cells--;
  }
}
```

The file `mapper.c` *is continued on the next page...*

C.16 Global Map Update Function

```
/*************************************************************************/
void update_global_map (struct local_map_ref *local_map,
struct global_map_ref *gmap)
/* Updating the global map is divided into two stages:
 * 1) Add cells from the local map into the global map
 * 2) Conglomerate the global map to condense data
 */
{
  int i,j,k; /*current cell in the local map*/
  float x_corner, y_corner, z_corner;  /*location of cell in the global map*/
  int index; /*index of the cell in the local map cell array*/
  float cell_size= local_map->cell_size;

  gmap->change_since_buffered= 1;  /*the map has changed since it
     was last buffered*/
  /*reinialize the map if size has been changed by user*/
  if (gmap->reinitialize == ON) {
    init_global_map (gmap);
    gmap->reinitialize= OFF;
  }

  /*take data from the local_map and congolmerate it into the global map*/
  index= 0;
  x_corner= local_map->x;
  for (i=0; i<local_map->x_cells; i++) {
    y_corner= local_map->y;
    for (j=0; j<local_map->y_cells; j++) {
      z_corner= local_map->z;
      for (k=0; k<local_map->z_cells; k++) {
global_map_add_cell (gmap, gmap->tree,
   x_corner, y_corner, z_corner,
   x_corner+cell_size, y_corner+cell_size, z_corner+cell_size,
   local_map->cells[index].occupied);
index++;
z_corner += cell_size;
      }
      y_corner += cell_size;
    }
    x_corner += cell_size;
  }

  /*conglomerate the global map as necessary*/
  conglomerate_tree (gmap, gmap->tree);
}
```

The file `mapper.c` *is continued on the next page...*

C.17 Mapping System Initialization and Update Functions

```
/*************************************************************************/
void init_mapper (struct mapper_ref *mapper,
  struct aavnavigation_ref *nav)
/* This function is called from within on_board.c to do all the
 * initialization for all the mapper routines (both local and global)
 *
 * In the current version, only the local stuff has been implemented.
 * Reinitialization is called from inside update_local_map and
 * update_global_map.
 */
{
  if (mapper->mapperIn->initialize == ON) {
    get_mapper_inputs (mapper->mapperIn, nav);
    init_local_map (mapper->mapperIn, mapper->local_map);
    init_global_map (mapper->global_map);
    mapper->mapperIn->initialize= OFF;
  }
}

/*************************************************************************/
void update_map (struct mapper_ref *mapper,
 struct aavnavigation_ref *nav)
/* This is the function that does it all.  It is called from within
 * the on_board.c code and makes the correct updates to the local and
 * global maps.
 */
{
  struct mapperIn_ref *in= mapper->mapperIn;
  struct local_map_ref *local_map= mapper->local_map;
  struct global_map_ref *gmap= mapper->global_map;

  get_mapper_inputs(in, nav);

  if (local_map->use_LocalMap) {
    update_local_map (in, local_map);
    gmap->update_count++;

    if ((gmap->use_GlobalMap) && (gmap->update_count == gmap->update_rate)){
      gmap->update_count= 0;     /*reset update counter*/
      update_global_map (local_map, gmap);
    }
  }
}
```

Appendix D

Mapping Structure

This appendix contains all the code that implements the mapping system structure for both the local map and global map. The mapping structure is defined by the file `mapper.spech`, which is included below. Section breaks have been added in this appendix to make the the code easier to read.

While effort has been made to ensure that this code is free of bugs, no guarantees are made about its correctness. In addition, it is likely that the current version of the `mapper.spech` file is different from the one included here.

This code is included as part of the `aav` version of the Draper simulation. The modified version of the `aav` simulation is stored in the directory `/spirit/disk5/people/rps1681/workarea/aav/`.

D.1 defines, includes, and headers

```
/*************************************************************************
 * Mapper.Spech
 *
 * Mapping software for the scanning laser rangefinder sensor that
 * will be added to the helicopter.
 */

/* We define mapper_types to avoid having this .spech file overwrite
 * itself if it is compiled again.*/
#ifndef __mapper_types__
#define __mapper_types__

\%include "switch.spech"
/* the definition for enumerated type Switch ??? */

\%define SWEEP_SAMPLES 9   /* Number of samples(ranges) per sweep */
/*Sweep size is designated using a constant so as to simplify sizing
 *arrays that store sweep results */
```

The file mapper.spec is continued on the next page...

D.2 Mapper Inputs

```
/*************************************************************************/
%Dir mapperIn_ref :Buffered inputs
/* stuff that comes from the hardware*/
{
| enum Switch   initialize    sw    ON    :Flag to initialize the map;

  /* heli position- constantly changing */
  double  heli_x       ft    0.0   :Heli position- north;
  double  heli_y       ft    0.0   :Heli position- east;
  double  heli_z       ft    0.0   :Heli position- down;
  double  heli_phi       rad   0.0   :Heli orientation - roll;
  double  heli_theta       rad   0.0   :Heli orientation - pitch;
  double  heli_psi       rad   0.0   :Heli orientation - yaw;

  /* sensor position on heli (currently doesn't change) */
  float sensor_x          ft          0    :sensor offset in heli frame of ref.;
  float sensor_y          ft          0    :sensor offset in heli frame of ref.;
  float sensor_z          ft          0    :sensor offset in heli frame of ref.;
  float sensor_phi        rad         0    :sensor orientation w.r.t. heli;
  float sensor_theta      rad         0    :sensor orientation w.r.t. heli;
  float sensor_psi        rad         0    :sensor orientation w.r.t. heli;

  /* scanner characteristics (shouldn't change) - mirrored in sensors.spech */
  float minRange   ft        0     :minimum measureable range;
  float maxRange   ft      100     :maximum mearureable range;
  float range_res         ft      0.25    :resolution of digital range counter;
  float angle0            rad    -0.4     :angle for count=0;
  float angle_res         rad     0.1     :resolution of angle measurement;
  float sweep_rate   na        5     :scan rate;
 |int   sweep_samples   na        SWEEP_SAMPLES  :data points in single scan;

  /* input data , constantly changing */
  int     range_counts[SWEEP_SAMPLES]   na   :range data in counts;
  int     angle_counts[SWEEP_SAMPLES]   na   :angle data in counts;
  float   ranges[SWEEP_SAMPLES]   ft   :measured ranges;
  float   angles[SWEEP_SAMPLES]   rad   :measured angles;
} mapperIn;
```

The file mapper.spec is continued on the next page...

D.3 Local Map Structure

```
/*********************************************************************/
typedef struct local_map_cell {
  int occupied;  /* high=occupied, low=open */
} local_map_cell;

/* for the sim frameworks (spec files) */
typedef   local_map_cell   *local_map_cell_array;
%type local_map_cell_array

/*********************************************************************/
%Dir local_map_ref      :heli-relative local map
/* The local map is a 3D grid representing the area nearby the helicopter.
 *
 * The local map is maintained relative to the helicopter itself.  As
 * the helicopter moves, the map moves as well.
 *
 * It is possible to change the size + resolution of the local map on the fly.
 * This is done by changing the appropriate size constants and then
 * flicking the resize switch.
 *
 * For the purpose of simplicity, all cells are cubes.
 */

{
  enum Switch   use_LocalMap  sw    ON    :Power control;
  double  x       ft   0.0  :Map origin position x=north;
  double  y       ft   0.0  :Map origin position y=east;
  double  z       ft   0.0  :Map origin position z=down;

  int x_cells                na   30 :Number of cells x=north;
  int y_cells                na   30 :Number of cells y=east;
  int z_cells                na   3  :Number of cells z=down;
  float cell_size           ft    4.0  :length of a cell edge;

  float x_desired_offset     ft    58  :desired offset of sensor in local map;
  float y_desired_offset     ft    58  :desired offset of sensor in local map;
  float z_desired_offset     ft    6   :desired offset of sensor in local map;
  float max_deviation        ft    6   :shift the map if this much deviation;
| float x_offset             ft    0   :current offset of sensor in local map;
| float y_offset             ft    0   :current offset of sensor in local map;
| float z_offset             ft    0   :current offset of sensor in local map;

  int hit_points              na    3   :cell increment for a hit;
  int miss_points             na    -1   :cell increment for a miss;

  enum Switch reinitialize   sw    OFF :flick switch to reinit the map;
| enum Switch reinit_buffer  sw    OFF :buffer still needs to be changed;
  local_map_cell_array    cells;  /*the cells in the map- see above*/
} local_map;
```

The file mapper.spec is continued on the next page...

D.4 Global Map Structure

```
/*************************************************************************/
/* global mapping structure */
%undef NONE    /*used to substitue PAUSE for NONE*/
%enum dimension {X, Y, Z, NONE};
%enum occupancy {CLEAR, UNKOWN, FULL};

typedef struct global_map_cell {
  int occupied;/* high=occupied, low=open */
  float x0; /*first corner of the cell*/
  float y0; /*first corner of the cell*/
  float z0; /*first corner of the cell*/
  float x1; /*second corner of the cell*/
  float y1; /*second corner of the cell*/
  float z1; /*second corner of the cell*/
  enum dimension split;  /*split dimension to create children*/
  struct global_map_cell *parent;
  struct global_map_cell *child0;
  struct global_map_cell *child1;
} global_map_cell;

/* for the sim frameworks (spec files) */
typedef   global_map_cell   *global_map_cellptr;
%type global_map_cellptr

/*************************************************************************/
%Dir global_map_ref           :kd-tree global map
{
  enum Switch   use_GlobalMap sw    ON    :Power control;
  float init_x0       ft     -300 :initial global map cell corner;
  float init_y0       ft     -300 :initial global map cell corner;
  float init_z0       ft     -60 :initial global map cell corner;
  float init_x1       ft     300 :initial global map cell corner;
  float init_y1       ft     300 :initial global map cell corner;
  float init_z1       ft       5 :initial global map cell corner;
  int init_occupied   na       0 :original global map assumption;

  int update_rate     na      4 :number of local updates per global update;
  int update_count    na      0 :number of local updates so far;
  float min_cell_size ft      5 :global map min cell size;
  int merge_tolerance na      3 :tolerance for conglomeration;
| int total_cells     na      0 :the number of cells in the global map;
  int change_since_buffered  na  1 :has the map changed since last drawn?;

  enum Switch reinitialize   sw    OFF :flick switch to reinit the map;
  global_map_cellptr tree;  /*the cells in the map- see above*/
} global_map;
```

The file mapper.spec is continued on the next page...

D.5 The Top-Level Structure

```
/*****************************************************************/
%Dir mapper_ref          :laser scanner mapping
{
  Dir (struct mapperIn_ref)     mapperIn    {mapperIn_dir};
  Dir (struct local_map_ref)    local_map   {local_map_dir};
  Dir (struct global_map_ref)   global_map  {global_map_dir};
} mapper;

#endif /*from the ifndef mapper_types */
```

Bibliography

[1] Guide to draper simulation framework.

[2] *Merriam Webster's Collegiate Dictionary (10th Edition).* Merriam-Webster, 1998.

[3] L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[4] J[ohann] Borenstein and Y[oram] Koren. Real-time obstacle avoidance for fast mobile robots. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(5):1179–1187, October 1989. shows that you don't have to draw the entire sonar cone.

[5] J[ohann] Borenstein and Y[oram] Koren. Histogramic in-motion mapping for mobile robot obstacle avoidance. *IEEE Transactions on Robotics and Automation*, 7(4):535–539, August 1991. demonstration of certainty grid working well in 2d robot.

[6] C. I. Connolly. Cumulative generation of octree models from range data. *Proceedings, International Conference on Robotics*, pages 25–32, March 1984. sample use of an octree.

[7] D. Eberly, R. Gardner, B. Morse, S. Pizer, and C. Scharlach. Ridges for image analysis. *Journal of Mathematical Imaging and Vision*, 4(4):353–373, December 1994. finding edges of objects is not easy.

[8] A. Elfes and L. Matthies. Sensor integration for robot navigation: cominbing sonar and range data in a grid-based representation. *Proceedings of the 26th IEEE Conference on Decision and Control*, pages 1802–1807, December 1987. discovery of occupancy grids?

[9] A[lberto] Elfes. Sonar-based real-world mapping and navigation. *IEEE Journal of Robotics and Automation*, RA-3(3):249–265, June 1987. abstractions in different levels of mapping - local+global.

[10] Donald Hearn and M. Pauline Baker, editors. *Computer Graphics, C Version.* Prentice Hall, New Jersey, second edition, 1997. computer graphics textbook.

[11] T[u] Jilin, D[ing] Mingyue, Z[hou] Chenping, and A[i] Haojun. Study of fast 3-d route planning approach for air vehicle. *SPIE*, 3087.

[12] Andrew E. Johnson and Martial Hebert. Seafloor map generation for autonomous underwater vehicle navigation. In *Autonomous Robots*, number 3, pages 145–168. Kluwer Academic Publishers, The Netherlands, 1996. an elevation map.

[13] A. Li and G. Crebbin. Octree encoding of objects from range images. *Pattern Recognition*, 27(5):727–739, May 1994. sample use of an octree.

[14] W[illie] Lim. Small spatial maps for mobile robots. *SPIE Mobile Robots*, 2352(9):116–127, 1994. an object list representation.

[15] H. P. Moravec. Sensor fusion in certainty grids for mobile robots. In A. Casals, editor, *Sensor Devices and Systems for Robotics*, number F52 in NATO ASI Series, pages 253–276. Springer-Verlag, Berlin, 1989. great overview of certainty grids.

[16] H. P. Moravec and A[lberto] Elfes. High resolution maps from wide angle sonar. *IEEE Conference on Robotics and Automation*, pages 116–121, 1985.

[17] D[aniel] Pagac, E[duardo] M. Nebot, and H[ugh] Durrant-Whyte. An evidential approach to map-building for autonomous vehicles. *IEEE Transactions on Robotics and Automation*, 14(4):623–629, August 1998. cell update rules- dempster-shafer is better than bayesian.

[18] Long Phan. Collision avoidance via laser rangefinding. Master's thesis, Massachusetts Institute of Technology, 1999.

[19] G[ary] Shaffer and A[nthony] Stentz. Automated surveying of mines using a laser rangefinder. In *Symposium on Emerging Computer Technologies for the Minerals Industry*, pages 363–370, Littleton, CO, February 1993. Society for Mining, Metallurgy, and Exploration, Inc. This is a full ARTICLE entry.

[20] C.M. Smith, J.J. Leonard, A.A. Bennett, and C.Shaw. In *Feature-Based Concurrent Mapping and Localization for Autonomous Underwater Vehicles*, Halifax, Nova Scotia, Canada, October 1997.

[21] Steve Steiner. Mapping and sensor fusion for an autonomous vehicle. Master's thesis, Massachusetts Institute of Technology, 1995.

[22] W. Kenneth Stewart. Three-dimensional stochastic modeling using sonar sensing for undersea robotics. In *Autonomous Robots*, number 3, pages 121–143. Kluwer Academic Publishers, The Netherlands, 1996. an elevation map.

[23] Christian Trott. Electronics design for an autonomous helicopter. Master's thesis, Massachusetts Institute of Technology, 1997.

[24] S. T. Tuohy, J.J. Leonard, J.G. Bellingham, N.M. Patrikalakis, and C.Chryssostomidis. Map based navigation for autonomous underwater vehicles. *International Journal of Offshore and Polar Engineering*, 6(1):9–18, March 1996.

[25] R[onda] Venkateswarlu. Multi-mode image based navigation for unmanned aerial vehicle. *SPIE Navigation and Control Technologies for Unmanned Systems*, 2738, 1996.