INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9111038

The use of a laser imaging system for automated vehicle guidance and space servicing tasks

Wu, Chris Kung, Ph.D.

Rice University, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

NOTE TO USERS

THE ORIGINAL DOCUMENT RECEIVED BY U.M.I. CONTAINED PAGES WITH POOR PRINT. PAGES WERE FILMED AS RECEIVED.

THIS REPRODUCTION IS THE BEST AVAILABLE COPY.

RICE UNIVERSITY

THE USE OF A LASER IMAGING SYSTEM FOR AUTOMATED VEHICLE GUIDANCE AND SPACE SERVICING TASKS

by

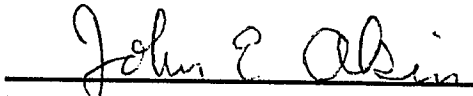
CHRIS K. WU

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENT FOR THE DEGREE

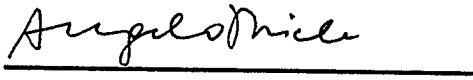
DOCTOR OF PHILOSOPHY

APPROVED, THESIS COMMITTEE:

Dr. John B. Cheatham, Director
Professor of Mechanical Engineering



Dr. John E. Akin, Chairman
Professor of Mechanical Engineering



Dr. Angelo Miele
Professor of Mechanical Engineering

Dr. Joe D. Warren
Asst. Professor of Computer Science

Houston, Texas
April, 1990

Abstract

The Use of A Laser Imaging System for
Automated Vehicle Guidance and Space Servicing Tasks

Chris K. Wu

This thesis presents a simulation of a laser imaging system for automated vehicle guidance and space servicing tasks. Data generated by this system are used to command a robot navigating in an unconstrained natural terrain. A computer graphics simulation program which emulates laser range scanners was developed and used for generating the required range images in this research. This simulation eliminates the need for an expensive laser scanner for developing laser data processing algorithms. Because the precise geometric information of the artificial objects in the scene is available, the simulation program can also be used as a testbed for evaluating laser data processing algorithms. Techniques and algorithms for converting laser range data to terrain maps were investigated and are presented in this thesis. A noise insensitive edge detector was developed for extracting wireframes of objects from the range data. By combining this edge detector with the Locus algorithm developed at Carnegie Mellon University, topographical terrain maps were derived from the sensory data. A 3-D sensor-based direct search algorithm was developed for finding collision-free optimal navigation paths using the terrain maps. This path planner generates paths of any desired degree of smoothness which can be readily used by the steering mechanism of mobile robots. A procedure for using a laser range finder and infrared proximity sensor for locating a satellite in space is also presented. The ultimate goal of this research is to develop a real time laser imaging system for automated vehicle guidance and object recognition.

Acknowledgements

I extend my gratitude to my principal adviser Dr. John B. Cheatham for all the assistance he has given me throughout the period I worked on my dissertation; to Dr. Rui J. P. de Figueiredo for his support and technical advice; and to Dr. Timothy F. Cleghorn of the Johnson Space Center for all of his help and guidance. I also thank the members of my committee, Dr. John E. Akin, Dr. Angelo Miele, and Dr. Joe D. Warren for their assistance in this effort.

I appreciate the support of this work by NASA/JSC under RICIS Research Activity A1.2 contract NCC-9-16.

A special thanks goes to my family and friends, who have been patient and supportive through the years.

Table of Contents

Chapter 1. Thesis Overview

1.1 Introduction 1

1.2 Thesis outline 2

Chapter 2. Introduction to Optical Vision and Laser Imaging System

2.1 Optical Vision System 7

2.2 Active Image Formation Systems 9

2.3 Active Imaging Techniques 10

2.4 Laser Range Scanner 11

2.5 Structure of a Laser Imaging System 12

2.6 State-of-the-Art Laser Imaging Systems 14

2.7 Conclusions 16

Chapter 3. Laser Imaging Simulation Algorithm (LISA)

3.1 Why a Simulation Program? 18

3.2 Comparisons of Z-buffer Data and Laser Images 19

3.3 Time-of-Flight Laser Scanners 21

3.4 World Modeling 26

3.5 Laser Imaging as A Spherical Perspective Projection Process 28

3.6 Cartesian Projection 30

3.7 View Plane Mapping 33

3.8 Calculating Range and Reflectance Data 36

3.9 Hidden Surface Removal 38

3.10 Noise in the Range and Reflectance Data 39

3.11 Ambiguity Interval 40

3.12 User Interface 41

3.13 Experimental Results 43

3.14 Conclusions 47

Chapter 4. Laser Imaging Processing for Generating Terrain Maps
4.1 Introduction 48
4.2 Cartesian Terrain Maps 49
4.3 Scrolling Procedure 52
4.4 Conversion to a Cartesian Representation 54
4.5 Framing the Image 56
4.6 Locus Algorithm 57
4.7 Range Shadows 60
4.8 Edge Detection 61
4.9 A Noise Insensitive Edge Detector 62
4.10 Detection of Erroneous Range Measurements Due to Weak Returned Signal 72
4.11 Conclusions 74

Chapter 5. Sensor-based Direct Search Algorithm (SDSA) for Autonomous Natural Terrain Navigation
5.1 Introduction 76
5.2 Natural Terrain Navigation 77
5.3 Mathematical Formulation 80
5.4 Direct Search Algorithm 83
5.5 Minimum Distance Collision-free Path Planning 85
5.6 Experimental Results 87
5.7 Collision-Free Path Planning for Multiple Mobile Robots 89
5.9 Conclusions 92

Chapter 6. Applications
6.1 Introduction 93
6.2 Rice-obot I 95
6.3 The Autonomous Navigation System 97
6.4 Application to Autonomous Indoor Navigation 99
6.5 Battery Replacement in a Satellite Model 102
6.6 Conclusion 107

Chapter 7. Summary and Conclusions
7.1 Summary 109
7.2 Conclusions and Outlook for the Future Research 112

References 115

Appendix A. 3-D Perspective Projection
A.1 Introduction 123
A.2 Normalization 124
A.3 3-D Clipping Against A Canonical View Volume 129
A.4 Inverse Scaling 132
A.5 Perspective Projection 133

Appendix B. Fitting a plane to Noisy Data by the Eigenvector Method
B.1 Fitting a Plane to Noisy Data 134
B.2 Probability Density Function of the Minimum Squared Error 136

Appendix C. Computer Program Listing 142

Chapter 1

Thesis Overview

1.1 Introduction

Laser imaging systems are used for making informed decisions via interpretation of laser data in order to accomplish assigned tasks with a certain level of intelligence and autonomy. In general, a laser imaging system consists of three modules: sensing, perception, and reasoning as shown in Figure 1-1. The sensing module includes a scanning laser range finder (or laser scanner) and control hardware. In some cases, a monocular television camera and sonar sensors are included as auxiliary sensors. This module is used primarily for translating the physical properties of objects in the robot environment into meaningful information required to accomplished the assigned functions [Nitzan 88]. Examples of the physical properties are geometric properties (e.g., position, orientation, volume, and shape) and surface reflectance. The perception and reasoning modules are composed of mainly software algorithms. Meaningful data are retrieved from the sensory data by the perception module and analyzed by the reasoning module. A plan of action is then initiated by the reasoning module in response to the appointed task.

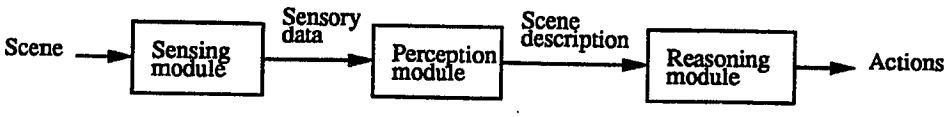


Figure 1-1. Generic model of a laser imaging system.

Laser imaging systems have proven useful in their ability to readily determine spatial relationships within an environment [Nitzan 77, Zuk 85]. Intelligent robotic applications such as automated mobile guidance and scene analysis have been successfully implemented by employing the 3-D ranging capability of laser imaging systems [Stenstrom 86, Daily 88]. State-of-the-art laser scanners provide not only high resolution range images within a fraction of a second, but also accuracy within a desirable range. It is certain that the development of laser data processing techniques is becoming the focus of research because of its wide range of potential applications in robotics.

1.2 Thesis outline

The objective of this research is to develop a computer simulation of a laser imaging system for 3-D automated vehicle guidance and space servicing tasks. The issues addressed in this thesis consist of four parts: the development of a computer graphics program to emulate laser scanners, techniques to transform and interpret range data into meaningful terrain maps, a 3-D navigation planner, and the use of auxiliary sensors such as the infrared proximity sensor for proximity operations. Chapter 2 of this thesis presents an introduction to laser imaging systems. State-of-the-art laser imaging systems developed at other research institutions will also be discussed in Chapter 2.

In developing laser data processing algorithms, the first difficulty which arises is the need for a laser scanner for generating required range data. Unfortunately, the cost of an industrial-grade laser scanner is extremely high. The ERIM (Environmental Research Institute of Michigan) and Odetics laser scanners,

for instance, cost well above $100,000 which puts them out of reach of many researchers. The lack of an affordable laser scanner has imposed a major restriction on developments in laser data processing algorithms. In order to bypass the hardware difficulty, a computer graphics program, called LISA (Laser Imaging Simulation Algorithm), was developed in this research to emulate laser scanners. LISA generates simulated laser images of various laser scanners. In addition, graphical models of objects can be combined and rearranged to create many different desirable scenes without the cost of actually manipulating the physical environment. LISA is implemented in 'C' language on an IBM-AT personal computer. This algorithm provides an easy and fast access to Computer-Generated (C-G) laser images. Moreover, since precise geometric information of the objects in the scene is also available, LISA can be used as a testbed for developing laser data processing algorithms [Wu 90]. Several algorithms developed at Rice University for autonomous robotic navigation [Cheatham 89a] have been designed and tested using the C-G laser images generated by LISA. A detailed discussion on the computer simulation program will be presented in Chapter 3.

Motion planning for mobile robots is becoming an important topic in robotics. In exploratory missions on another planet, for example, unmanned vehicles will be used to traverse uncharted areas and collect environmental samples. Since the remote distance prevents instantaneous communication between the vehicles and human operators, local autonomy for planning a safe path on a possibly rough terrain is demanded. There are three subtasks involved in the automated vehicle guidance problem: obtaining topographical terrain maps from the sensor data

(sensing), finding a safe, smooth path using the terrain maps (perception and reasoning), and determining the desirable motion along the path (action). The last subtask is extremely important if multiple robots are navigating in the same environment.

Currently, the proposed navigation planners have either been constrained to the flat ground assumption [Daily 87], or produce only a line-arc-line type of path [Nelson 89]. Very few planners even consider the motion problem [Kant 87]. For most applications, steering mechanisms are required to maintain the position and heading of mobile robots to continuously align with the path. Since instantaneous changes in direction at the discontinuous transition points of the arc-line-arc paths are physically impossible, errors in position and heading of mobile robots are inevitable. A better navigation planner which generates a smooth path on a natural (not necessary flat) terrain is needed.

To achieve the goal, a 3-D natural terrain navigation system was developed and is described in this thesis. A noise insensitive edge detector was implemented for extracting wireframes of objects from range data. By combining this edge detector with the Locus algorithm developed at Carnegie Mellon University [Hebert 89], topographical terrain maps were transformed and interpolated from the range data. Regions in the terrain maps are classified into scanned, unscanned, and occluded areas. A 3-D Sensor-based Direct Search Algorithm (SDSA) was developed in this research for finding the optimal navigation paths using the terrain maps. The SDSA algorithm generates paths of any desired degree of smoothness which can be readily used by the steering mechanism of mobile robots. Gradient

and elevation tests were employed to check the traversability of every candidate path. The significance of this navigation system is twofold. First, it is designed for natural terrain navigation. Second, since no obstacle detection is employed in the system (collision-free path is ensured by the traversability test), the processing time is minimum. Chapter 4 will address the processing techniques for generating terrain maps using the range data. Mathematical formalism and software implementation of the SDSA algorithm will be presented in Chapter 5. Chapter 6 of this thesis illustrates the use of this 3-D navigation system for maneuvering a mobile robot developed at Rice [deFigueiredo 89] in an environment with obstacles randomly placed.

In space servicing tasks such as replacing modular components for satellites, precise location and orientation of the satellites must be determined. At a nominal Space Station altitude of 270 nautical miles, the sunlight intensity will fluctuate between about sixty minutes of extreme brightness and thirty minutes of nearly darkness [Bronez 87]. Because of the extreme illumination conditions in space, special lighting techniques and comprehensive image enhancement algorithms will be needed if an optical vision system is chosen for the tasks [Krishen 87]. Researchers in this area are now investigating the prospects of using a laser imaging system as an alternative system. Chapter 6 proposes a framework for locating satellites using a laser imaging system and proximity sensors. A simplified case in which a laser range finder (Lasernet by Namco) and infrared proximity sensors are used to located a satellite model is also discussed in Chapter 6.

The last chapter of this thesis summarizes this research. Future research as extension of this study is also discussed.

Chapter 2

Introduction to Optical Vision and Laser Imaging System

2.1 Optical Vision System

A vision system is one of the most important elements that enables a robot to do intelligent tasks [Besl 85, Kanade 87]. The primary function of a vision system is to provide three dimensional (3-D) world models of the current state of the robot environment via interpretation of sensor data. To date, robotic systems with vision capability have successfully illustrated their abilities to interact with a dynamic environment to achieve assigned tasks with some level of intelligence [Maravec 83, Flynn 88, Matthies 88, Sorensen 89].

In the 1970's and early 1980's, vision systems were implemented by analyzing the intensity images (pictures) of the scene taken by a monocular television camera. Systems of this types are referred to as monocular vision systems. The problems emphasized in the literature during those years were techniques and algorithms for extracting object properties from intensity pictures [Duda 73, Rosenfeld 82, Gonzalez 87, Horn 87]. As a result of these studies, researchers realized three major problems inherent to 2-D picture processing techniques. First, most of the techniques require massive computation time which make monocular vision systems impractical for real-time applications. Second, uncontrollable uncertainties associated with the intensity images dominate the reliability of those techniques. Some examples of those uncertainties are the illumination effects such as shading, shadows, and specular reflections [Ballard 82].

Finally, the lack of providing 3-D information constrains monocular vision systems to limited applications. Because of these deficiencies, monocular vision systems have lately been judged of limited utility. Nevertheless, the numerous picture processing techniques developed for monocular vision systems have built a solid foundation for robotic vision systems.

In recent years, research efforts have branched out and have started to emphasize techniques for extracting 3-D information. Also better hardware has become available to speed up the processing time. To date, various techniques have been developed for extracting the depth information of a scene from multiple intensity images. Binocular stereo vision [Maravec 83, Matthies 88], shape from shading [Ikeuchi 81], range from focus [Krotkov 86], and photometric stereo vision [Horn 86] are examples of some well known techniques. These types of vision systems will be referred to as stereo vision systems. Good surveys on stereo vision systems can be found in the papers by Besl and Jain [Besl 85] and Kanade [Kanade 87]. Although stereo imaging techniques overcome the deficiency of monocular vision systems, the cost is the additional computation time (for extracting the depth information) added to an already slow process. Moreover, these techniques are still sensitive to external illumination.

With regards to the hardware improvements, many new hardware architectures have been proposed as solutions for fast computation. Some examples are the concepts of distributed computing and parallel processing. Unfortunately, the hardware and software implementations of these concepts is still far from maturity. The problems of slowness and uncontrollable reliability in processing

intensity images are still the major barriers for applying optical vision systems for practical applications.

2.2 Active Image Formation Systems

Another approach to solving the vision problem is to use the range data produced by an active image formation system such as a range finder to eliminate the necessity of conducting intensive shape analysis procedures that seek to produce the 3-D information. This idea was first implemented in 1971 by Shirai and Suwa [Shirai 71]. The range finder used in their experiments employed a vertical slit projector and a TV camera. The slit projector emits a sheet of light whose direction is controlled by a scanning mirror. A light strip resulting from intersection of the light sheet with objects in the field of view is observed by a TV camera. The distance from the range finder to each sampled point on the light strip is calculated by triangulation with the known positions of the scanning mirror and the TV camera. By rotating the mirror from left to right, many points in the field of view are obtained. The active image formation system provides two very promising features: direct range measurements and independence from external illumination. Active imaging systems have attracted considerable attention from researchers because they appear to be the most promising solution of 3-D vision problems. After Shirai and Suwa's paper was published, basic research in processing range images has progressed in two directions. The first is concerned with developments of the best hardware for measuring range data and the second is to exploit laser data processing techniques for various applications.

2.3 Active Imaging Techniques

There have been many active image formation systems proposed and implemented over the years. Among those systems, ultrasonic and radio wave range finders have been studied the most [Drumheller 85, Elfes 86, Parnis 89]. Despite low cost and simplicity, these two systems do not provide the high resolution that is required in many robotic applications. Another technique, the structured light method, utilizes a controlled light source and a TV camera. 3-D information of the scene is determined by observing the light pattern projected on objects [Asada 87, Gordon 87, Asada 88]. The fundamental principles of this type of system are similar to the system developed by Shirai and Suwa. This type of system is categorized as a triangulation vision system. If the light source is a laser beam, the system will then be referred to laser triangulation system. There are two advantages of this type of system. First, it produces range images of any desirable resolution [Asada 88]. Elevation data of the scene sampled uniformly along the x and y axes are determined by triangulating the known light source and the camera positions. Second, since a TV camera is used for observing the light patterns, conventional picture processing techniques can be readily applied. However, there are also two shortcomings associated with triangulation vision systems. The first is the problem of "missing data". Since the triangulation method is used to compute the range data, accuracy of the range measurements depends on the separation distance (disparities) between the camera and the light projector. Large disparities result in missing data because many points in the scene will be seen by the light projector but not the camera. The second problem is due to the fact that conventional picture

processing techniques are time consuming, which results in a slow range data acquisition rate. The speed problem can be improved by replacing the TV camera with a 2-D array of discrete photo-detectors. This approach was demonstrated by Araki *et al* [Araki 88]. The patterns of the projected light can be quickly obtained by scanning the photo-detectors at high speed. Nevertheless, the "missing data" problem remains unsolved.

2.4 Laser Range Scanner

In 1977, Nitzan et al. [Nitzan 77] described an experimental time-of-flight laser scanner that generated range and reflectance images. A HeNe laser is used to transmit a laser beam to a target object and the reflected beam was detected by a receiver. The radial distance is determined by measuring the time needed for the laser beam to travel from the transmitter to the object and back. The reflectance value is measured proportional to the strength of the detected returned signal. A scanning mirror is used to direct the laser beam along the pan/tilt orientations with equal angular increments so that dense samples of the field of view are obtained. This system has successfully demonstrated the feasibility of producing a 128 x 128 x 8 range image at a distance of 1-5 meters with a range accuracy of about 1 cm. Moreover, since the laser detector is arranged in such a way that the received light is coaxial with the transmitted beam, see Figure 2-1, the missing data problem that is inherent to triangulation sensing is eliminated. In recent years, laser range scanner technology has progressed extensively as diode laser technology has advanced. The laser scanner designed by Environmental Research Institute of

Michigan (ERIM), for example, generates a 64 x 256 x 8 range image every 0.5 seconds [Zuk 85]. The range accuracy is 3 inches at a distance of 64 feet. Horizontal and vertical fields of view are 80 and 30 degrees, respectively.

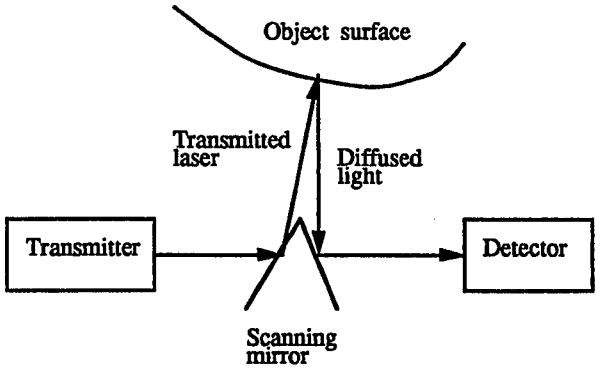


Figure 2-1. Detected laser is coaxial with the transmitted laser.

2.5 Structure of A Laser Imaging System

As discussed in the previous chapter, a laser imaging system consists of a laser scanner, control hardware, and application algorithms. It is designed for making informed decisions via interpretation of laser data in order to accomplish assigned tasks with a certain level of intelligence and autonomy. Its three modules (sensing, perception, and reasoning) can be arranged in either an open loop (see Figure 2-2a), or a feedback loop structure (see Figure 2-2b). With the open loop structure, these modules are executed sequentially for every execution loop. This structure is commonly adapted in applications where the observed environment is

well structured. In the parts sorting problem, for example, the operation sequence is well defined [Svetkoff 84]. An open loop structure is adequate for this type of application. The advantages of the open loop structure are that it is easy to implement and the straightforward process makes hardware implementation of the system feasible. With the feedback loop structure, local intelligent units are attached to each of the three modules. If undesirable phenomena are detected by the local intelligent process, iterative operations of the corresponding module may be performed. Interactions between local intelligent units are also permitted so that modular operations can be altered by the decisions of other intelligent units. For example, in some space applications the laser imaging system is used for locating a satellite navigating in space. Iterative sensing will be commanded if the sensing intelligent unit detects no returned signal. The information will be passed to other intelligent units so that no further evaluation of the range data will be performed. Because of the local intelligence, the feedback loop laser vision system provides better performance and adaptability to a time-varying environment.

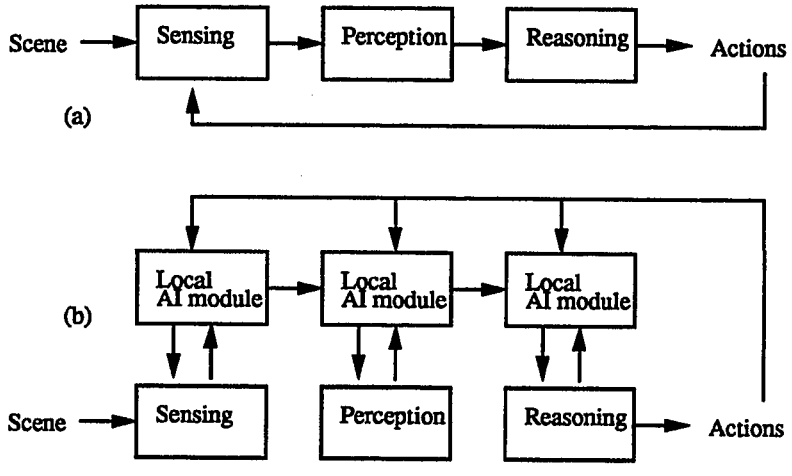


Figure 2-2. Laser imaging system arranged in (a) open loop (b) closed loop structure.

2.6 State-of-the-art Laser Imaging Systems

Several state-of-the-art laser imaging systems have been developed for automated vehicle guidance and scene analysis. At the Hughes AI center, a laser imaging system for cross-country navigation was developed and tested in natural terrain in 1987 [Daily 88]. The objective of the laser imaging system was to produce a traversability map of the robot environment for navigation path planning. An ERIM laser range scanner was used for obtaining the terrain information. A segmented elevation map was obtained by fitting surface patches to the range data. A *spring model* of the vehicle was used to test each surface patch for traversability and label it accordingly [Daily 87]. A simple navigation strategy which finds straight line segments from the traversable regions was used to obtain the navigation path. This navigation system provides reasonable performance on natural terrain but generates paths consisting of straight line segments. Since instantaneous changes in direction are physically impossible, errors in position and heading of the mobile robots at the discontinuous points on the path are inevitable. To alleviate the situation, Kant and Zucker [Kant 87] proposed a post-processing technique for smoothing the line segments. Nevertheless, the smoothed path contains B-spline segments rather than a global parametric equation that describes the entire path.

A similar system designed for on-road navigation was developed at Carnegie-Mellon University and Martin Marietta Space System, [Goto 87, Dunlay 88, Thorpe 88]. An additional color camera was added to the system. Road boundaries were determined from the color images based on the assumption that road color is distinguishable from that of sides. The road boundaries were then mapped to the

elevation map. Searching for obstacles and path planning were performed only in the area inside the road boundaries. Since the road was assumed to be a flat surface, an obstacle was defined as a region in the elevation map in which at least a predefined number of points projected above or below the road surface. A 10 cell Warp machine (100 MFLOPS super computer) was used for color image processing. This system has successfully maneuvered an autonomous land vehicle (ALV) around obstacles while remaining on the road at speeds up to 8 km/hr. Yet this system has not demonstrated it ability to navigate on a natural terrain.

The University of Maryland [Asada 88] developed a simple map building system for a mobile robot. This system uses a structured light scanner and a monochrome camera as the sensing devices. This system produces height maps (similar to the segmented elevation map) from the sensor data. The height maps contain regions that are labeled as unexplored, occluded, traversable and obstacle regions. The unexplored regions are simply the spaces outside the field of view of the range finder. Occluded regions represent regions of the workspace occluded by obstacles. Since the assumption of flat road surface was used, traversable regions were defined as the points on the height map near the assumed ground plane. Other regions were then classified as obstacle regions. A merging process was used for expanding the traversable regions by merging other surrounding points which have low slope and low curvature. The major problems of this system, as discussed in the preceding sections, are the massive computations needed and the inherent "missing point" problem. In addition, the accuracy of the structured light method depends strongly on the surface geometry, reflectivity of viewed objects and the

condition of the ambient light.

At General Electric Corporate R & D [Stenstrom 86], a laser vision system which is capable of reconstructing wireframe models of polyhedra from multiple range views of a scene was developed. A triangulation laser range finder was used to obtain range data. For each range image, points of high curvature are determined, grown into connected components, and then converted into line segments. These line segments are then jointed to form a wire-frame for each view. The polyhedra was obtained by intersecting the wireframes extracted from each view.

2.7 Conclusion

The primary function of a vision system is to provide meaningful world information of the current state of the robot environment via interpretation of sensor data. Among these vision systems, laser imaging systems have proven to be the most effective. A laser imaging system consists of a laser scanner, control hardware, and application algorithms. It is designed for making informed decisions via interpretation of laser data in order to accomplish assigned tasks with a certain level of intelligence and autonomy. Laser scanners possess the ability of readily measuring range information, which greatly reduces the processing time required for analyzing sensing data. Their independence from external illumination also provides reliable performance in various environments. State-of-the-art laser scanners provide not only high resolution range images within a fraction of a second, but also accuracy within a desirable range.

Several state-of-the-art laser imaging systems have been built and tested successfully for automated vehicle guidance and scene analysis. However, since most navigation algorithms employ a flat earth assumption, applications of these systems are constrained to 2-D robot environments. For many applications, such as under-water mining and exploratory missions on other planets, it is required to navigate on rough terrain. Although the Hughes navigation system was designed for natural terrain navigation, it generates paths consisting straight line segments which are not desirable for smooth navigation. In addition, considerable processing time is required to generate the surface patches for the terrain maps. Since natural terrain navigation is becoming an important practice in robotics, research efforts are continue on developing a better 3-D navigation system.

Chapter 3

Laser Imaging Simulation Algorithm (LISA)

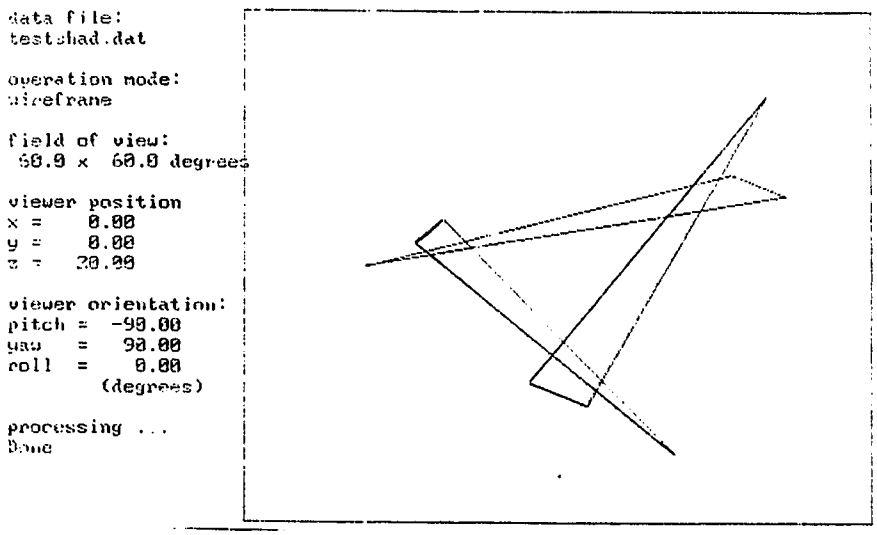
3.1 Why a Simulation Program ?

The preceding chapter described the wide range of potential applications of laser imaging systems. There are also several survey papers on applying range data for object recognition [Besl 85, Brady 88]. Development of laser data processing techniques is becoming the focus of research in the areas of computer vision and robotics. In developing laser data processing algorithms, the first difficulty is the need for a laser scanner for generating required laser data. Unfortunately, a state-of-the-art laser scanner is very costly. For example, the ERIM and Odetics laser scanners cost well above $100,000 dollars, putting them out of the reach of many researchers. Modeling and maneuvering a physical environment for desired scenes are also costly and time-consuming. Safety consideration also prevents the wide spread use of laser scanners. Safety precaution is extremely important for operating high power laser equipment. For these reasons, many laser imaging algorithms have been developed and tested on computer generated (C-G) range images [Besl 85]. Computer graphics provide not only fast and easy access to laser range data, but also the opportunity for studying the limitation of laser imaging systems without encountering hardware difficulties.

3.2 Comparisons of Z-buffer Data and Laser Images

The C-G range images can be created by extracting the Z-buffer data (depth buffer) from a standard 3-D computer graphics system. However, there are two major differences between the Z-buffer data and the actual laser range data. First, the Z-buffer data correspond to an orthogonal projection of the 3-D objects on a 2-D rectangular image plane as shown in Figure 3-1. The range data, on the other hand, correspond to the radial measurements of the distance in a spherical coordinate system. The orthogonal projection guarantees a similarity transformation while the range data does not. Straight lines and geometric shapes, for examples, are preserved in the Z-buffer data but not in the laser range data. Secondly, the field of view observed by laser ranging devices has a hyperbolic shape (depicted in Figure 3-2). The computer graphics techniques, in contrast, have a pyramid field of view. In addition to these differences, a good C-G laser image should also display the shortcomings of the real laser image such as the wrap around interval, missing points, and noise.

To generate more realistic C-G laser images, a Laser Imaging Simulation Algorithm (LISA) was designed and presented in this chapter. LISA imitates precisely the laser images using the system parameters of the simulated laser scanner entered by the user. All parameters are changeable on the simulation program to include hardware specifications, output data size, and viewing parameters. Both the ERIM and Odetics laser scanners, as well as future scanners, can be emulated. From the input parameters, LISA calculates several system characteristics such as the signal-noise-ratio, ambiguity interval of the scanners and



(a)

(b)

Figure 3-1. (a) Orthogonal projection of triangles interlacing each other, (b) corresponding laser data. The gray scale of the image is proportional to the range value. Lighter shades represent greater value.

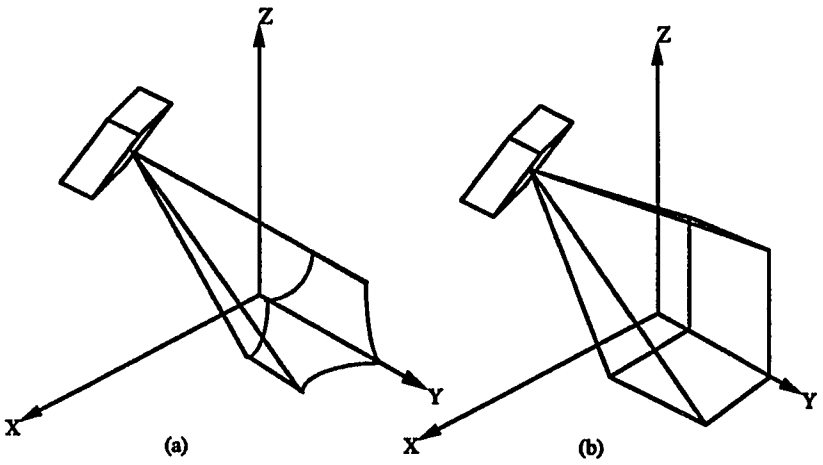


Figure 3-2. Field of view of (a) Scanning laser range finder

(b) standard camera

viewing volume and then generates the appropriate laser images.

3.3 Time-of-Flight Laser Scanners

A review of the laser scanner will aid the discussion of the simulation program. As shown in Figure 3-3, a time-of-flight scanning laser range finder consists of four components: transmitter, scanning mirror mechanism, photodetector, and comparator unit [Nitzan 77, Jarvis 83]. A time-of-flight laser system measures distance by determining the time needed for the laser to travel from the transmitter to the target and back. That is,

$$r = \frac{(t * c)}{2} \qquad (3.1)$$

where

r is the distance measured from the laser scanner to objects

t is the total traveling time

c is the speed of light constant

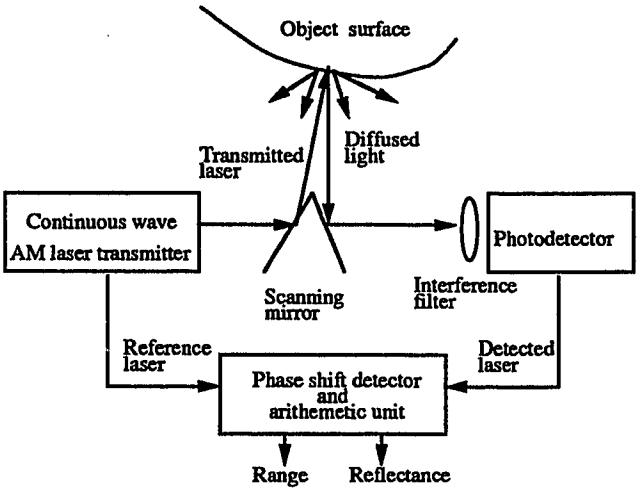


Figure 3-3. Generic Model of a time-of-flight laser scanner.

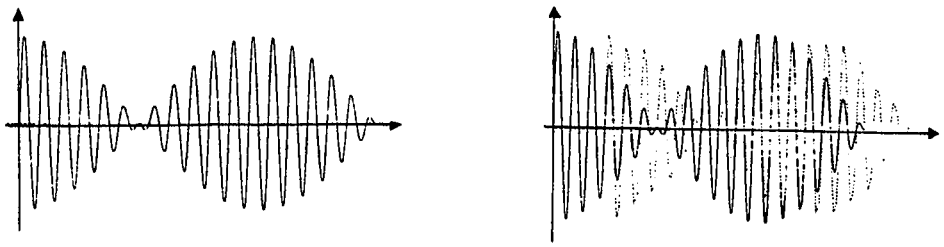
Two methods, the elapsed time and phase shift methods, are commonly used for measuring the traveling time t. The elapsed time method directly measures t by utilizing an electronic timer, which is triggered by the transmitted laser and stopped by the returned laser. This method is suitable for long range measurements, where t is relatively large and therefore can be measured with reasonable accuracy. The phase shift method compares the phase difference of the transmitted and returned laser and then calculates the distance accordingly. The transmitted laser is amplitude modulated (AM) as depicted in Figure 3-4a. By comparing the transmitted and received laser as shown in Figure 3-4b, the phase difference is measured and the distance r is calculated which is proportional to the phase

difference. The relation between the phase difference p_d and the range r is given by,

$$r = \frac{p_d}{2} \frac{c}{2\pi f_m} \tag{3.2}$$

where

f_m is the modulation frequency (Hz) and $p_d \in [0,2\pi)$



(a) (b)

Figure 3-4. (a) Continuous wave amplitude modulated laser, (b) phase difference of the transmitted (solid) and returned (dashed) laser.

There is one problem with the phase shift method. If the phase difference is a multiple of 2π, the two signals will overlap and the hardware returns a phase difference of zero. This means that the distance returned is actually the remainder of the true distance divided by one wavelength of the modulation frequency (called ambiguity interval) as shown in Figure 3-5. Because of this problem, the phase shift method is commonly used for short range measurements, usually within one or two ambiguity intervals. In general, the phase shift method provides better range

accuracy than the elapsed time method because the phase shift detector can obtain higher accuracy, usually within a tenth or even hundredths of a degree.

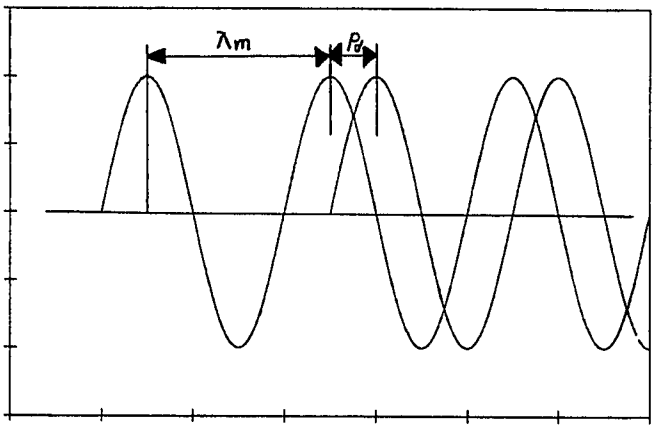


Figure 3-5. Phase difference measured is the remainder of the actual distance divided by the modulation wavelength.

As shown in Figure 3-3, the transmitter generates either continuous wave AM laser or pulsed laser, depending on which method is used for range measurements. The transmitted laser is deflected to a target by the scanning mirror and the diffused laser is detected by the photodetector. A reference laser generated by the transmitter and the detected laser are sent to a comparator which calculates the range value. The comparator also generates a reflectance value which is proportional to the strength of the returned laser.

The field of view of the scanning laser has a hyperbolic shape defined by the total horizontal and vertical scanning angles, depicted in Figure 3-2b. Let Θ and

Ψ represent the total horizontal and vertical scanning angles and the pair (θ,ψ) represent a specific laser direction. By using stepper motors for controlling the position of the scanning mirror, a polar gridwork having N_c horizontal divisions and N_r vertical divisions is generated as shown in Figure 3-6. At each grid point, range and reflectance data are measured. Two rectangular arrays are used to store the range and reflectance measurements. Let the pair (r,c) represent the row and column position in the array. The relations between the laser direction (θ,ψ) and the array element (r,c) are given by,

$$\theta = \Theta \left(\frac{c}{N_c} - \frac{1}{2} \right)$$

$$\psi = \Psi \left(\frac{r}{N_r} - \frac{1}{2} \right) \qquad (3.3)$$

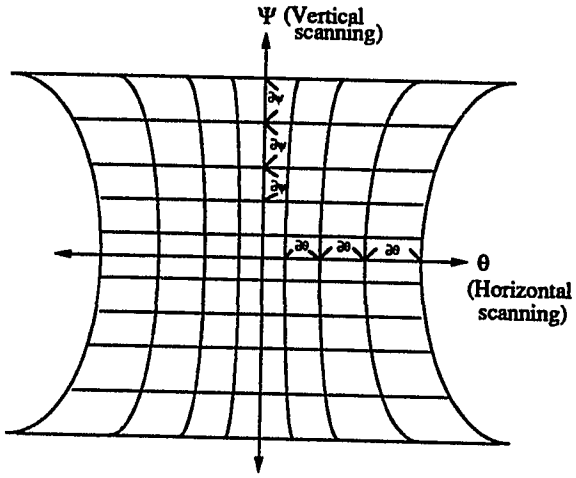


Figure 3-6. A polar gridwork generated by the scanning Laser.

Knowing the relations between the laser direction (θ,ψ) and the corresponding array element (r,c), it is then understood that both of the notations $r(\theta,\psi)$ and $r(r,c)$ represent the range measurement along the direction specified by (θ,ψ).

3.4 World Modeling

The first task in developing LISA was to decide on the representation method for modeling the artificial robot environment. World modeling techniques can be classified into two categories: models for pattern recognition (computer vision) and models for scene construction (computer graphics). For clarity, let *object modeling* represent the modeling techniques designed for pattern recognition and *solid modeling* represent the modeling techniques designed for scene construction. In general, solid modeling emphasizes the quantitative properties (the surface boundaries, volumes, and spatial locations) and object modeling emphasizes the qualitative properties (the shapes, colors, and mathematical descriptions) of the objects. For example, if a soccer ball and a cubic box are the objects to be represented, solid modeling methods will model the ball with a sphere equation and location of its center and the cubic box by the locations of its eight vertices. For recognition purpose, only the distinguishable features of the objects are significant. Therefore, the cubic box will be identified as an object with six vertices; the soccer ball will be identified as one without a vertex, provided that there is a clear definition of *vertex*. Since LISA is a computer graphics program, the solid modeling techniques will be suitable.

Many solid modeling techniques now exist. Constructed solid geometry,

octree representations, wireframe representation, boundary surface representation, and sweep techniques are some of the well-known techniques [Allen 82, Requicha 82]. Among these modeling techniques, the surface boundary representation (SBR) method was used in the simulation program. This method is selected for its simplicity and large number of library models available. Object models can be generated and combined easily into any desired scene. The SBR method models objects by defining the 3-D planar polygons that bound the objects. Each of the polygon is defined by the locations of their vertices, and a topology list describing the connectivities of these vertices. Arbitrary surfaces are approximated to any desired degree of accuracy by utilizing many polygons. Figure 3-7 shows the data file of a cubic box. This method is popular because the model surface area and volume are well defined. In addition, all object operations are carried out using planar algorithms [Newman 79, Foley 82, Harrington 83, Salman 87].

no. of vertices no. of surfaces	8 6
X_{min} X_{max} Y_{min} Y_{max} Z_{min} Z_{max} = box enclosing the object	0 1 0 1 0 1
X Y X : coordinates of each vertex	0 0 1 0 1 1 1 1 1 1 0 1 0 1 0 1 1 0 1 0 0 0 0 0
no. of vertices in each surface every vertex index on each surface	4 1 2 3 4 4 2 3 6 5 4 5 6 7 8 4 1 4 7 8 4 3 4 7 6 4 1 2 5 8

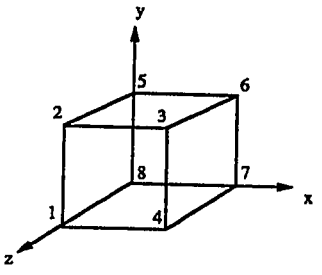


Figure 3-7. Surface boundary representation of a cubic box.

3-5 Laser Imaging as A Spherical Perspective Projection Process

The laser image formation process discussed in section 3.3 can also be interpreted as a transformation of 3-D objects in the spherical world (r,θ,ψ) onto a 2-D projection plane as depicted in Figure 3-8. The projection plane is a cross section of the view volume consisting of a polar gridwork of N_c horizontal divisions and N_r vertical divisions. It is perpendicular to the center line (at which θ and ψ are both equal to 0) of the view volume.

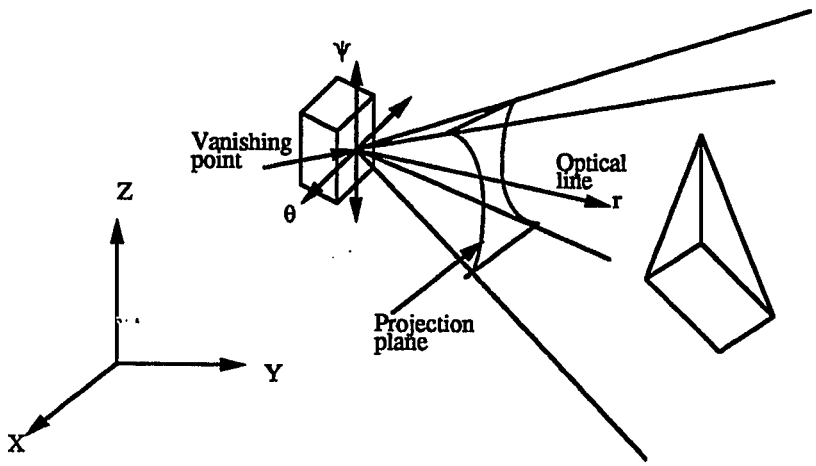


Figure 3-8. Transformation of 3-D object onto a 2-D projection plane.

Let the term *optical line* represent the center line and the term *view plane* represent the projection plane of the view volume. By treating the center of the scanner mirror (where the optical line passes through) as the "vanishing point" (or view point) of the projection, the laser image formation process can be perceived as a spherical perspective projection of 3-D objects onto the 2-D polar gridwork. The

value at each grid point (denoted by $r(\theta,\psi)$) represents the distance from the vanishing point to the nearest object along the direction specified by the grid point location. For simplicity, let the terms *spherical projection* and *Cartesian projection* represent perspective projection in spherical coordinates and a Cartesian coordinates, respectively. Many techniques have been developed for Cartesian projection [Foley 82, Hearn 86]. Unfortunately, direct conversions of these algorithms to spherical coordinates is not trivial. The difficulties are twofold. First, artificial environments can be easily defined in a Cartesian space but not in a spherical one. Second, coordinate transformations from Cartesian to spherical are nonlinear. Nevertheless, spherical projection can be achieved by performing the following steps:

(1) Perform a Cartesian projection onto a Cartesian view plane. The Cartesian view plane is a cross section of the pyramid view volume bounded by the four corners of the spherical view plane as shown in Figure 3-9. A left-handed Cartesian coordinate system is attached to the view point with the z-axis aligned with the optical line. Objects inside the pyramid view volume is projected to the view plane using a standard Cartesian projection technique.

(2) Overlay the polar gridwork onto the Cartesian view plane and find all grid points which intersect with the projected objects. Each of the intersecting point indicates that the scanning laser intersects the objects in the field of view along the direction specified by the grid point. This step is referred as the process of view plane mapping.

(3) Calculate the range and reflectance value at each of the intersecting

grid point.

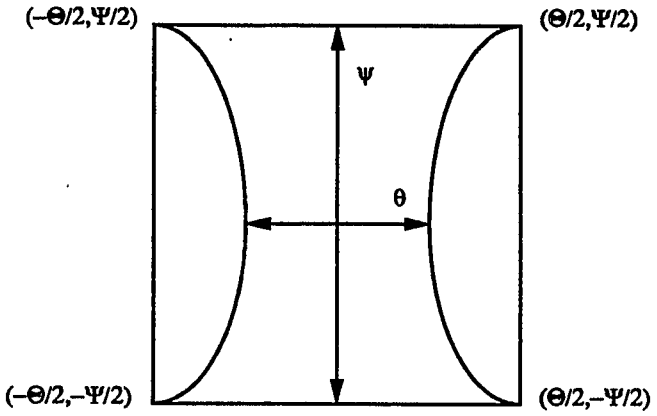


Figure 3-9. Cartesian view plane determined by the spherical projection plane.

The block diagram of the implementation of the spherical projection is shown in Figure 3-10. To illustrate the spherical projection process, let's consider a simple case of generating the laser image of a polygon that is partial inside the view volume as depicted in Figure 3-11.

3.6 Cartesian Projection

Given the specifications of a view volume, Cartesian projection is done by first clipping the objects against the view volume and then projecting the clipped objects onto the view plane. Clipping is a procedure for eliminating all parts of

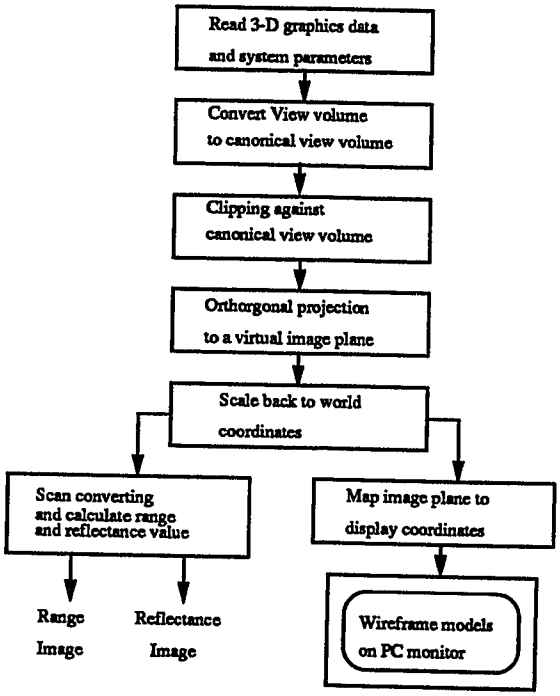


Figure 3-10. Block diagram of the spherical projection.

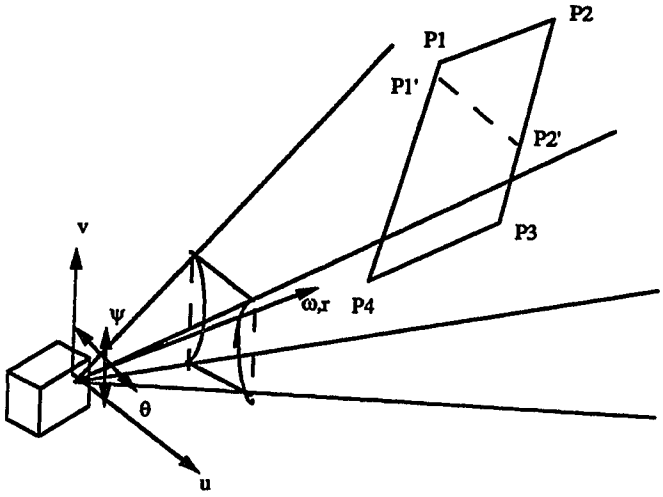


Figure 3-11. Spherical projection of a polygon.

objects outside the view volume. It can be done by calculating the intersections of the objects with each of the boundary planes of the view volume. However, the large number of calculations of this process calls for considerable computing. Fortunately, there are view volumes that are easier to clip. For example, the canonical view volume (as depicted in Figure 3-12) defined by the five planes,

$$z = 0,\ x = z,\ x = -z,\ y = z,\ y = -z,$$

requires a minimum number of calculation.

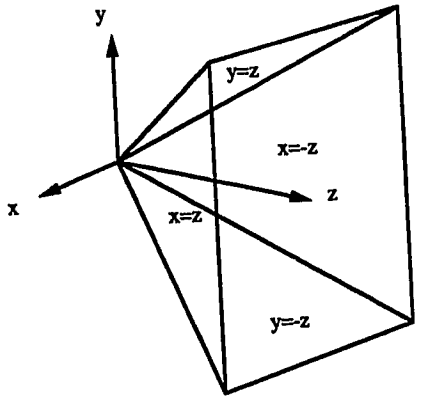


Figure 3-12. A canonical view volume.

Whether a point in space is inside or outside of the view volume can be determined by simply comparing its coordinate components. For example, the point is above the view volume if its y-component has a value greater than its z-component. To take the advantage of clipping in the canonical view volume, the arbitrary view volume is transformed into a canonical one before performing the clipping process. This transformation is called the normalization process. As shown in Figure 3-13,

the Cartesian projection is implemented in four steps: normalization, clipping, scaling, and perspective projection. The scaling procedure is performed after the clipping process to restore the relative distances distorted by the normalization process. Details of the techniques and algorithms used for Cartesian projection are discussed in Appendix A.

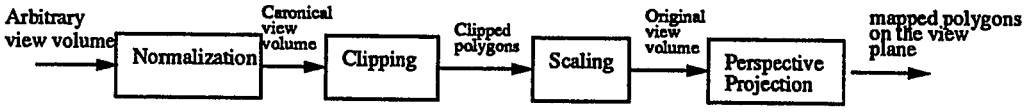


Figure 3-13. Block diagram of Cartesian projection.

3.7 View Plane Mapping

After the polygon was normalized and projected onto the Cartesian view plane, the next step is to perform view plane mapping. As shown in Figure 3-6, the spherical view plan consists of a polar gridwork of N_c horizontal divisions and Nr vertical divisions. Each of the grid points corresponds to a specific laser direction. The task of view plane mapping is to find all grid points on the spherical view plane that intersect with the projected polygon on the Cartesian view plane. Let u-v-w denote the left-handed coordinates attached to view point as depicted in Figure 3-14. The v coordinate of each scan line on the Cartesian view plane is given by

$$v = d \times \tan(\psi) \qquad (3.4)$$

where

$\psi \in [\ -\Psi/2, \Psi/2\ ]$

d is the w coordinate of the view plane.

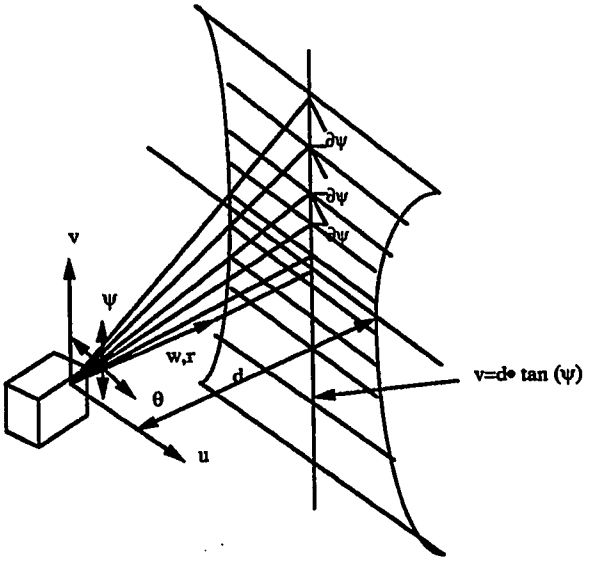


Figure 3-14. v coordinates of the scan lines on the Cartesian view plane.

For small Ψ, say $|\Psi| \leqslant \pi/3$, equation 4 can be approximated by

$$v = d \times \psi \qquad (3.5)$$

For each scan line, the intersections of the scan line with all edges (called edge points) of the polygon are determined. Spherical coordinates of those edge points are then calculated using the following equations,

$$\psi = \tan^{-1}(v/d)$$

$$\theta = \sin^{-1}(u/l) \qquad (3.6)$$

where $l = (u^2 + v^2 + d^2)^{1/2}$

As shown in Figure 3-15, all grid points between a pair of edge points intersect the projected polygon. Let (θ_s,ψ) and (θ_p,ψ) represent the spherical coordinates of a pair of edge points s and p, respectively. The spherical coordinates of intersecting grid points between them are given by

$$(\theta_s+k\delta\theta,\psi),\ k = 0, 1, 2, ..., (\theta_s-\theta_p)/\delta\theta \qquad (3.7)$$

where $\delta\theta = \Theta/N_c$.

If the edge points are outside of the spherical view plane, that is $|\psi| > \Psi/2$ (see Figure 3-15), they will be adjusted to generate the intersecting points correctly.

The procedure described above for finding intersecting points is analogous to the scan converting algorithm developed for polygon filling in the computer

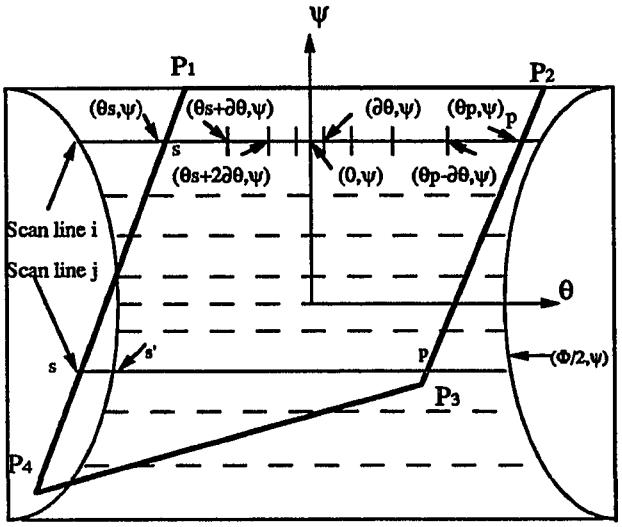


Figure 3-15. Scan converting to find all grid points that intersect the projected polygon. Notice that the edge point s on scan j is outside the spherical projection plane, s' will be used for determining the intersecting points.

graphics. Foley and Van Dam introduced an edge coherence and scan line algorithm that is adapted by the simulation program. With a little modification, this algorithm was used to find all intersecting points in the simulation program. Details of the edge coherence and scan line algorithm can be found in the book by Foley and Van Dam [Foley 82].

3.8 Calculating Range and Reflectance Data

Knowing that the laser intersects with the polygon along a specific direction (θ,ψ) as shown in figure 3-16, helps one to determine u and v coordinates of the intersecting point by,

$$u = w\frac{\tan(\theta)}{\cos(\psi)}$$

$$v = w\tan(\psi) \qquad (3.8)$$

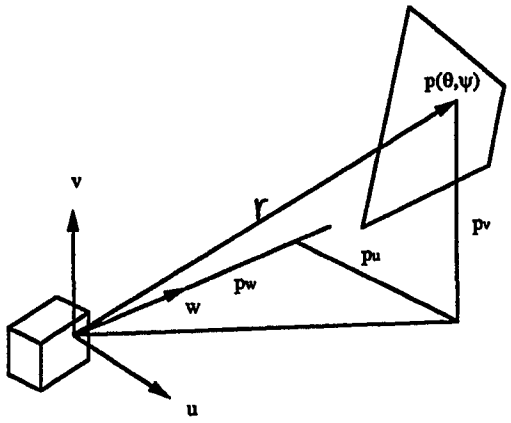


Figure 3-16. Cartesian coordinates of a intersecting point.

To determine the w-coordinate of the point, the plane equation of the polygon is used. As shown in Figure 3-17, let { $(u_1\ v_1\ w_1)$, $(u_2\ v_2\ w_2)$, $(u_3\ v_3\ w_3)$, $(u_4\ v_4\ w_4)$ } denote the normalized coordinates of the polygon vertices, the plane equation of the polygon can be calculated from the first three vertices. That is

$$Au+Bv+Cw+D=0$$

where $A = v_1(w_2-w_3) + v_2(w_3-w_1) + v_3(w_1-w_2)$
$B = u_1(w_3-w_2) + u_2(w_1-w_3) + u_3(w_2-w_1)$
$C = u_1(v_2-v_3) + u_2(v_3-v_1) + u_3(v_1-v_2)$
$D = u_1(w_2v_3-v_2w_3) + u_2(v_1w_3-w_1v_3) + u_3(w_1v_2-v_1w_2)$

Substituting u and v into the plane equation, one will get

$$w = \frac{-D}{(A\tan(\theta)/\cos(\psi) + B\tan(\psi) + C)} \qquad (3.9)$$

The range value can then be obtained by

$$r(\theta,\psi) = (u^2 + v^2 + w^2)^{1/2} \qquad (3.10)$$

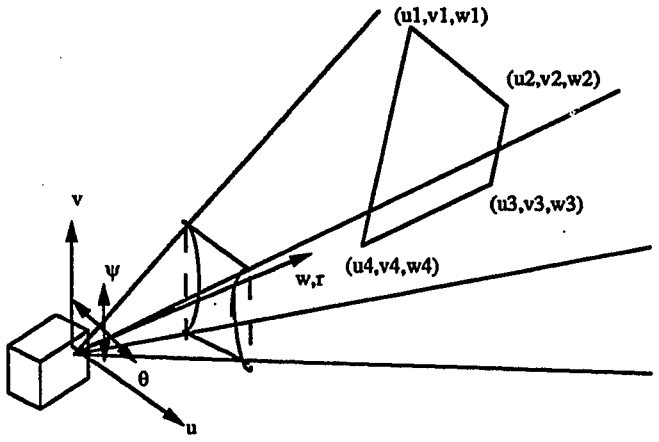


Figure 3-17. Normalized polygon.

The reflectance value is a function of the strength of the laser energy, reflectivity of the surface, and surface geometry relative to the laser scanner. Since surface properties are not provided by the SBR method, the reflectance value can not be computed unless the surface properties are known. In the simulation program, however, the polygons are assumed to be Lambersian surfaces. The reflectance of a Lambersian surface is proportional to the cosine of the angle between the surface normal and the incident light, that is

$$i(\theta,\psi) = \alpha(\mathbf{l}\cdot\mathbf{n}) \qquad (3.11)$$

where

α is the surface reflectance constant ($\alpha \in [0,1]$)

$\mathbf{l}$ is the unit vector along (θ,ψ)

$\mathbf{n}$ is the surface normal of the polygon

In general, the Lambersian surface is a good approximation to many man-made substances [Moon 61, Wendlandt 66]. It is also the most commonly used assumption in developing digital image processing techniques.

3.9 Hidden Surface removal

With the SBR representation, the artificial environment is composed of (or approximated by) many planar polygon. Processing each polygon in turn can produce a laser image of a scene. Two memory buffers: a depth-buffer and a reflectance-buffer are used to stored the results. The depth-buffer holds the range

data of the visible surfaces in the field of view. The reflectance-buffer stores the reflectance information of the corresponding surfaces in the depth-buffer. An algorithm similar to the z-buffer algorithm was used to determine the visible surfaces. During the scan converting process, the following steps are performed for every intersection point (θ,ψ)

1) Calculate $r(\theta,\psi)$ and its corresponding array location (r,c) in the depth-buffer.

2) If r(r,c) is less than the depth-buffer value at (r,c), then

   (a) place r(r,c) into the depth-buffer and

   (b) place corresponding reflectance value into the reflectance-buffer.

When the condition in step 2 is true, the polygon point is closer to the viewer, new range and reflectance are recorded.

3.10 Noise in the Range and Reflectance Data

In a real laser system, many sources of noise degrade the range and reflectance measurements. Some examples include transmission noise, photon noise generated by the photodetector, ambient noise, and noise in the subsequent amplifiers. To simulate noise, LISA generates artificial noise based on the theory derived by Nitzan *et. al.* [Nitzan 77]. Nitzan's theory states that the range measurement error generated by an amplitude-modulated phase-shift laser rangefinder can be modeled as a normally distributed random noise with the standard deviation

$$\sigma_r = \frac{1}{2\sqrt{2}\,\pi}\,\frac{\lambda_m}{m\ \mathrm{SNR}}\ r$$

$$\mathrm{SNR} = \left(\frac{\alpha\eta\lambda A_R F_T T}{\pi hc}\ \rho_d \cos(\theta)\right)^{1/2} \qquad (3.12)$$

where λ_m is the modulation wavelength,

m is the modulation factor,

SNR is the signal-noise-ratio,

α is the interference filter constant ($\alpha \in [0,1]$),

η is the quantum efficiency of the photodetector ($\eta \in [0,1]$),

λ and F_T are the wavelength and average power of the laser,

A_R is capture area of the photodetector,

T is the average sampling time,

ρ_d is the diffuse constant of the object surface ($\rho_d \in [0,1]$),

θ is the angle between the incident light and the surface normal,

h and c are the Plank's and speed of light constants.

The derivation of this noise was based on a general model of an AM laser scanner. Experimental data obtained from the AM laser scanner developed at Stanford Research Institute revealed a general agreement with the computed results [Nitzan 78].

3.11 Ambiguity interval

After generating the range and reflectance data of a scene, the ambiguity interval effect is simulated by dividing each range value by the ambiguity interval

and storing the remainder back to the depth-buffer. The depth- and reflectance-buffer are then combined and saved to a file for further processing. The output data is arranged in the standard 16-bit format. The higher 8-bit stores the range data; the lower 8-bit stores its corresponding reflectance value.

3.12 User Interface

LISA features a user friendly interface. System parameters of the simulated laser scanner are requested using the following menu driven screens:

Screen 1:

Enter the name of the data file [default demo.dat] ?
Select the operation mode
(1) wireframe
(2) wireframe + range
Your choice [default 2]?
Do you want to add noise to the laser image [default yes] ?
Enter your instrument parameters:
wavelength of the laser beam (mM) [default 632.8] ?
power of the laser beam (mW) [default 15.0] ?
modulation frequency (MHz) [default 9.0] ?
capture area of the receiver (M^2) [default 1.5e-4] ?
acquisition time per frame (sec) [default 0.8] ?
the signal-noise-ratio of your system is approximately
43.7 dB at a distance of 5 meters.
Is this computer equipped with an EG display [default yes] ?

Screen 2:

Enter the viewer location [default= 0, 0, 50]

x=

y=

z=

Enter the light source location [default= 0, 0, 0]

x=

y=

z=

Enter the viewer orientation [default = -90, 0, 0]

pitch=

yaw=

roll=

Note: "the light source location" is irrelevant for generating laser images. This is reserved for future use.

Screen 3:

Enter the horizontal scanning angle of the laser scanner

[default= 60 degrees]

theta =

Enter the number of pixels along the horizontal scanning line:

[default: 128 pixels]

number =

Enter the vertical scanning angle of the laser scanner

[default : 60 degrees]

theta=

Enter the number of pixels along the vertical scanning line

[default : 128 pixels]

number=

Enter the ambiguity interval [default=16.7M]

interval=

Two modes of operation are provided by LISA. In the wireframe mode, the wireframe models of the corresponding scene are generated and displayed on the PC monitor. To save computation time, hidden line removal is not performed for this wireframe display. This mode is primarily used for previewing the output images to obtain a desired viewer position. After the system parameters and viewer location have been set to desired values, LISA can be switched to a second operation mode to generate the C-G laser images.

Six viewing parameters are used to define the scanner position in space. The viewer location specifies the location of the view point (the center of the scanning mirror) and the viewer orientation defines the view volume in space. The ambiguity interval is determined by the modulation frequency. Nevertheless, the simulation program permits users to modify this value regardless of the modulation frequency.

3.13 Experimental Results

An artificial robot world consisting of a hex-shaped track is created for generating laser images as shown in Figure 3-18. It is bounded by a one foot high wall along the interior and a one foot deep ditch along the exterior. These boundaries test for the wandering limits for the robot. In a real application, these artificial obstacles could be established to direct the robot along a desired path. Several obstacles are randomly placed along the pathway. The range images generated in this thesis simulate the Odetics laser images. The horizontal and vertical fields of view of the odetics laser scanner are 60 by 60 degrees, and the

output data size is 128x128. The scene in wireframe (without hidden line removal) is displayed on the PC monitor (depicted in Figure 3-19). A corresponding range image is developed as shown in Figure 3-20a. Random noise characterized by the simulated laser scanner can also be added to the C-G laser data in Figure 3-20b. To illustrate how an curved object can be approximated by using planar polygons, Figure 3-21 shows the range image of a martini glass.

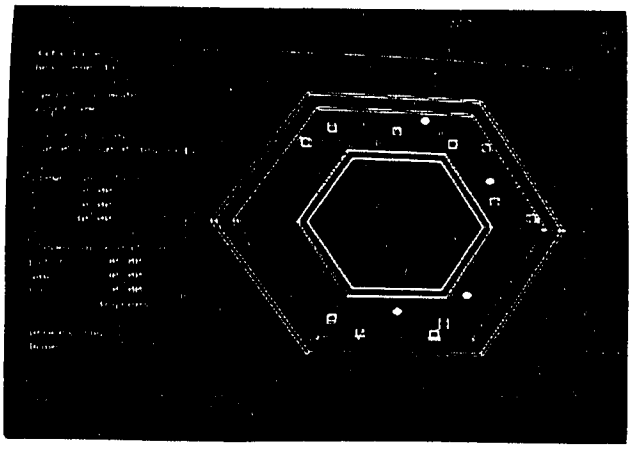


Figure 3-18. An artificial robot world.

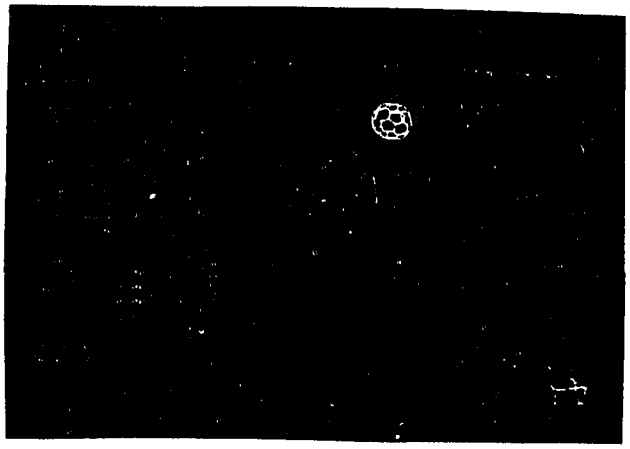
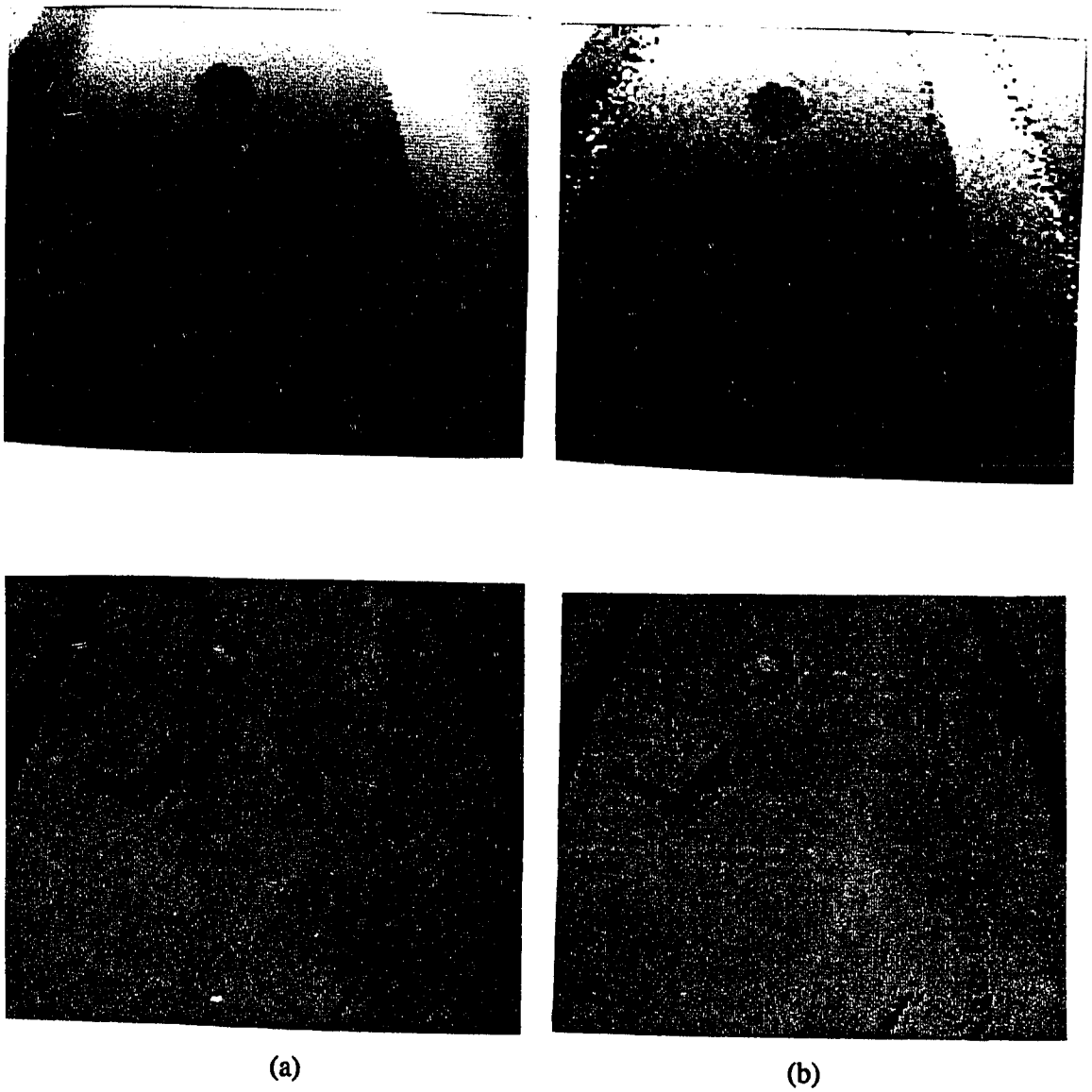


Figure 3-19. A view of the robot world.



(a) (b)

Figure 3-20. Range (top) and reflectance (bottom) data of scene in Figure 3-19. (a) noise free, (b) SNR = 15 dB.

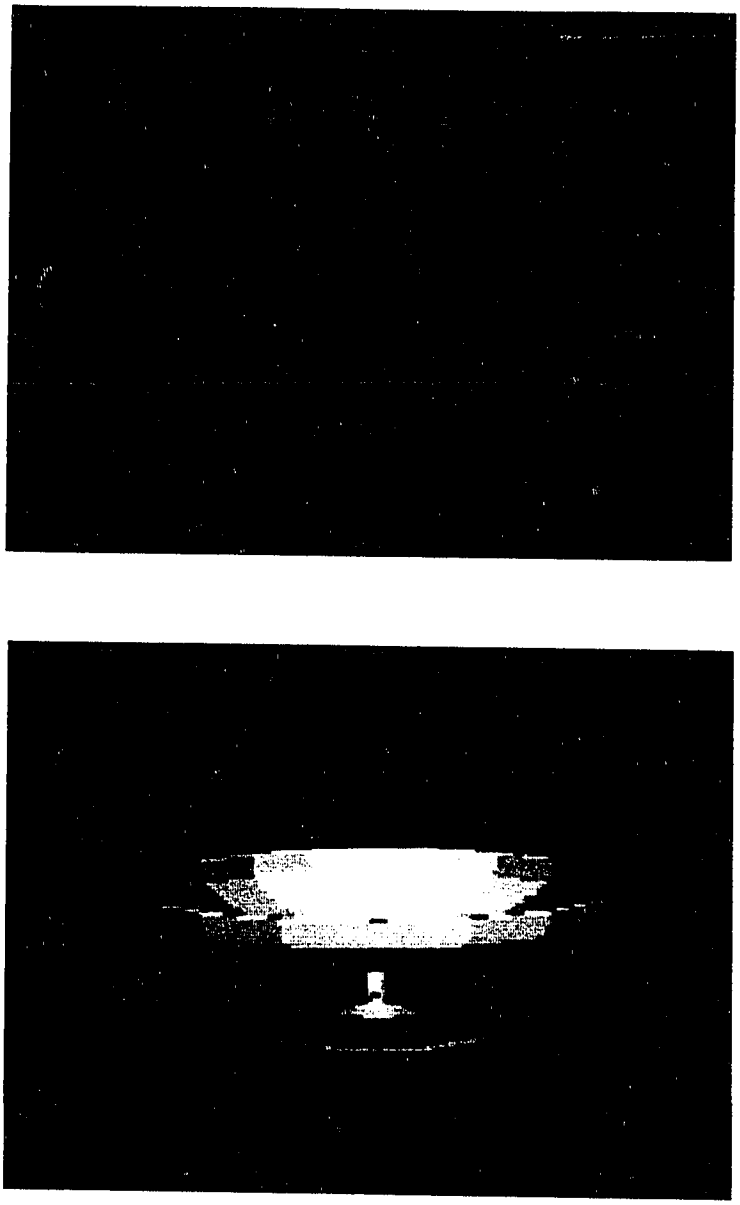


Figure 3-21. Range and reflectance data of a martini glass.

3.14 Conclusion

A Laser Imaging Simulation Algorithm (LISA) was developed to emulate laser scanners such as the ERIM and Odetics scanners. This system generates C-G laser range and reflectance data and displays system shortcomings of real laser images including wrap around interval, missing points and noise. These features are useful in developing algorithms for use with real laser imaging systems. LISA was implemented in C language on an IBM-AT. Source codes of this program are included in Appendix C. A Microsoft C compiler was used to compile the program. Wireframe models of the viewing objects are also generated by LISA and can be displayed on a PC monitor. Hidden line removal was not performed for the wireframe display in order to save computational time. The output range and reflectance data are arranged in standard 16-bit format. Output data are stored in a file that can be transported to any Vision Development Work-station for further processing.

Chapter 4

Laser Image Processing for Generating Terrain maps

4.1 Introduction

An objective of laser image processing is to transform and interpret laser data into meaningful environment information to support the robot's applications. In general, laser image processing algorithms fall into two categories: enhancement and perception. Enhancement algorithms improve the quality of the laser data. Examples include the scrolling procedure [Dunlay 86] for detecting the ambiguity interval phenomenon and the donut filter [Svetkoff 84] for improving noisy data. Perception algorithms are mainly task-oriented. For autonomous robotic navigation, they provide information about the surroundings to the path planner so the robot can steer safely through its environment.

This chapter describes the laser data processing techniques implemented at Rice [Cheatham 89]. The task is to generate a terrain map from a single laser data set. Since a mobile robot is a dynamic system, it must be able to reason about terrain information that is extracted from sensory data taken along its course of trajectory in order to interact with its environment. Some of the techniques described in this chapter were originally developed by other research organizations, therefore for detailed discussions of these techniques one should refer to the original papers.

4.2 Cartesian Terrain Maps

The main task of a perception system is to convert laser data into a terrain representation. Terrain representations are categorized into three areas: configuration space, graph, and grid. The configuration space (CS) represents obstacles (all non-traversable regions) by polyhedra enclosing them as shown in Figure 4-1. Its data base consists of Cartesian coordinates of polyhedra vertices and a topology list providing the connectivities of these vertices. The CS model was first introduced by Lozano-Perez [Lozano 83] as the solution for the *find-path* problem. A path is generated by connecting line segments between vertices unobstructed by obstacles. One can find the minimum distance path by using the length of the line segments as the cost criterion. With its simplicity and robustness, the CS model has become the most popular representation technique in robotics and artificial intelligence [Winston 84]. However, the CS model requires a large number of computations for finding obstacle regions and corresponding polyhedra from range data. Moreover, Oommen [Oommen 87] indicates that the application of this type of terrain map to 3-D navigation is questionable and impractical. Construction of irregular shape objects into 3-D models from range data is expensive and complex. For these reasons, configuration space is mainly used for applications involving in well-known, static environments.

Graph representations arrange position information of the objects and free spaces into a hierarchical structure as shown in Figure 4-2. The whole workspace is represented by its root node, and the root node has four (for 2-D applications) or eight (for 3-D applications) child nodes. Each child node corresponds to a

subregion of the workspace. If the subregions are free spaces or fully occupied by objects, the child nodes are labeled accordingly. If only a portion of a subregion is occupied by objects, the corresponding node will be labeled as a mix node, with four or eight child nodes that are labeled in the same fashion. Examples of the graph representation include the quadtree (2-D) and octree (3-D). Graphs provide an intuitively appealing interface to efficient path planning operations [Liu 89]. They also make matching sensor data with existing representations straightforward. Nevertheless, construction of hierarchical graphs from range data are complex and computationally expensive.

The grid representation is a direct representation of the robot environment. It is a rectangular gridwork representing a top-down view of the terrain. Each grid point represents an (x,y) location in space and its value represents the elevation (z) of the terrain at that point. For clarity, let the term *elevation map* denote the grid representation. An elevation map is an alternate representation of the range data. It

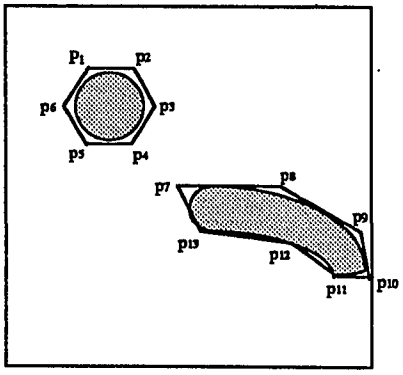


Figure 4-1. Objects defined in a 2-D configuration space.

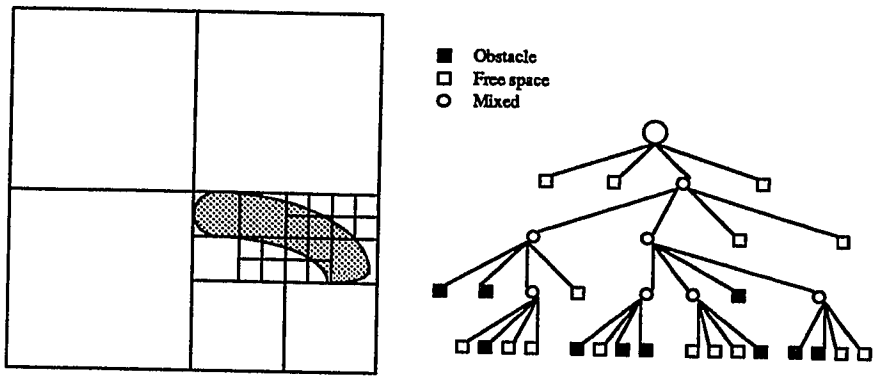


Figure 4-2. A 2-D graph representation of an object [Allen 82].

is obtained by converting laser data into Cartesian points and storing the results in the map. Since the coordinate transformation is a straightforward operation, elevation maps appear to be better for representing sensory data for real time navigation. There is one problem with elevation maps; maps resulting from direct coordinate conversions have non-uniform data distribution. Elevation data grow less dense and accurate with increasing distance from the sensor. To overcome this problem, Hebert *et. al.* [Hebert 89] developed the locus algorithm for interpolating range data to obtain a dense elevation map on a uniform grid.

This thesis selected the grid representation for its simplicity and its popularity among navigation planners. It also provides a better mechanism for combining spatial information from different sources. Maps built from different sensory data (e.g., laser, sonar, and TV camera) can be easily combined by adding (properly weighted) the map information together [Shafer 86].

4.3 Scrolling Procedure

The ambiguity interval phenomenon (AIP) must be detected and corrected from range data before an elevation map can be correctly created from them. Dunlay and Morgenthaler [Dunlay 86] proposed a scrolling procedure for detecting the AIP in the range data. Dunlay's algorithm states that the AIP can be detected by scrolling up the laser data row by row and looking for differences in column values exceeding a specified threshold. If the threshold is exceeded, the AIP is found and one ambiguity length is added to the top value. This algorithm assumes that the range measurements have finite changes in distance as the scanner moves one step further from the origin as depicted in Figure 4-3. A large change in range values indicates a discontinuity caused by the modulation. Experimental data provided by Dunlay indicated that this analysis works relatively well for the first two ambiguity intervals.

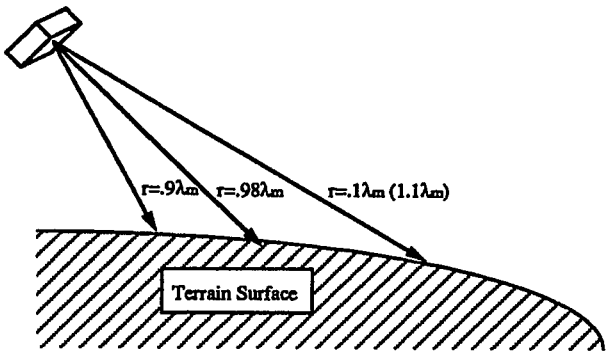


Figure 4-3. Range discontinuity due to ambiguity interval phenomenon

In this thesis, the laser scanner is assumed to be set up in such a way that the field of view of the scanner covers the area immediately in front of the mobile robot as shown in Figure 4-4. Using the center point (the robot's location) of the first scan line as a reference point, the algorithm scrolls outward to detect AIP across the first scan line. Once this is established, the algorithm scrolls up the laser image and checks for abnormal discontinuities between column range values. A threshold of 20 feet has been arbitrarily set in the program. Figure 4-5 shows how the range ambiguity appears in the range image.

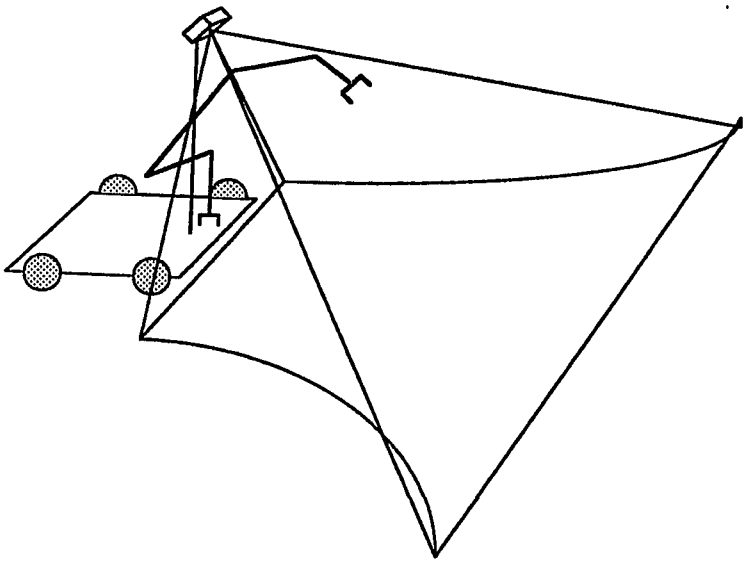


Figure 4-4. Field of view of a mobile robot.

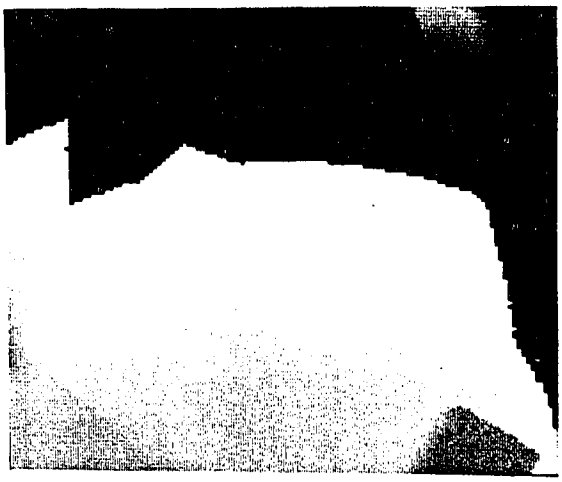


Figure 4-5. Range image contains ambiguity interval phenomenon.

4.4 Conversion to a Cartesian Representation

Transformations from spherical coordinates to Cartesian coordinates are given by

$$x(\theta,\psi) = r(\theta,\psi)\sin(\theta)$$

$$y(\theta,\psi) = r(\theta,\psi)\cos(\theta)\sin(\psi)$$

$$z(\theta,\psi) = r(\theta,\psi)\cos(\theta)\cos(\psi) \qquad (4.1)$$

Given the position (r,c) of the range data, the polar coordinates (θ and ψ) are calculated by Equation 3.3. Notice that θ and ψ given by Equation 3.3 are measured relative to the left-handed Cartesian coordinates (local coordinates)

attached to the scanner as depicted in Figure 4-6. Cartesian points calculated from Equation 4.1 are thus defined in the local coordinate system. Once the geometric relations between the world coordinates and the local coordinates are known, world coordinates of those points can be derived by simple coordinate transformations.

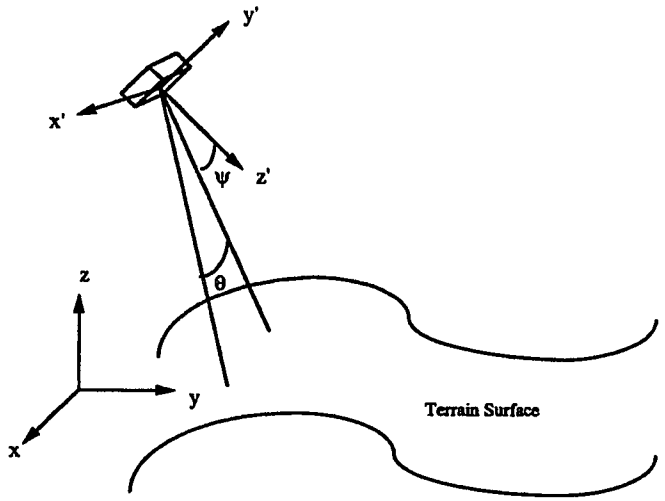


Figure 4-6 Polar angles are measured relative to the local coordinates.

By using Equation 4.1 and transforming local to world coordinates, each point in the range data is mapped to a Cartesian point. Elevation at that point corresponds to the z value. As shown in Figure 4-7, a converted image contains many data points having no elevation value. These data points are referred to as *missing points* (not to be confused with data points containing elevation value of zero). There are three types of missing points: points outside the field of view, points caused by the sparseness of laser data, and points in regions occluded by objects in the scene (Another type of missing point, the *invisible missing point*, will be introduced in section 3.10). Procedures for distinguishing and adjusting these missing points are discussed in the following sections.

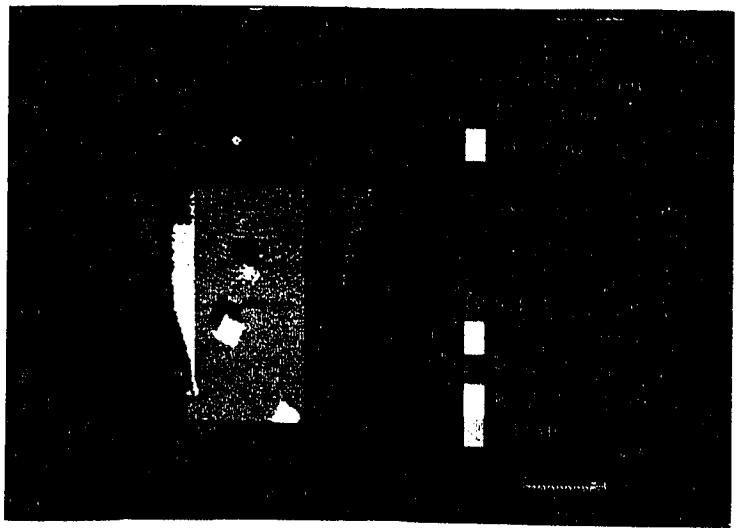


Figure 4-7. Elevation map calculated from Equation 4.1.

4.5 Framing the Image

In the elevation maps, regions outside the field of view of the scanner are labeled as *unscanned area.* A framing procedure developed by Weiland [Weiland 90] identifies missing points in the unscanned area by scanning the map from left to right, labeling all missing points as unscanned points until reaching the first non-missing point. Then the same procedure restarts on the other side of map. After scanning horizontal, vertical scanning will proceed in much the same way. Experimental results, shown in Figure 4-10, show the effectiveness of this procedure. However, missing points due to sparseness or occlusion on the edge of the terrain map will be mislabeled as unscanned. Since a navigation system would not chart a path into the unscanned and occluded area, this would not pose a problem.

4.6 Locus Algorithm

The locus algorithm [Hebert 89] developed at Carnegie Mellon University is an interpolation procedure to fill in the missing points caused by data point sparseness. The problem of finding the elevation of a point (x,y) is equivalent to find the intersection of a vertical line with the terrain surface. Since the surface is known by its sample points, analytical computation of the intersecting point is not possible. If the elevations of adjacent points are known, the elevation can be interpolated from its neighbors. However, if its neighbor points are also missing points, it is difficult to interpolate in the Cartesian space. The basic idea of the locus algorithm is to perform the interpolation in the image space (the spherical coordinates employed by the laser scanners). A vertical line is a locus (curve) in the laser image, whose equation as function of ψ is given by:

$$d(\psi) = \sqrt{\frac{y^2}{\cos^2(\psi)} + x^2}$$

$$\theta(\psi) = \tan^{-1}\left(\frac{x\cos(\psi)}{y}\right) \qquad (4.2)$$

The problem is then to find the intersecting point (if it exist) of the curve described by Equation 4.2 and the discrete surface in the range data. This is done by searching along the curve for the intersecting point. The search consists of two stages. As in Figure 4-8, the algorithm first searches for the two scan lines (ψ_1 and ψ_2) between which the surface intersects the curve are located.

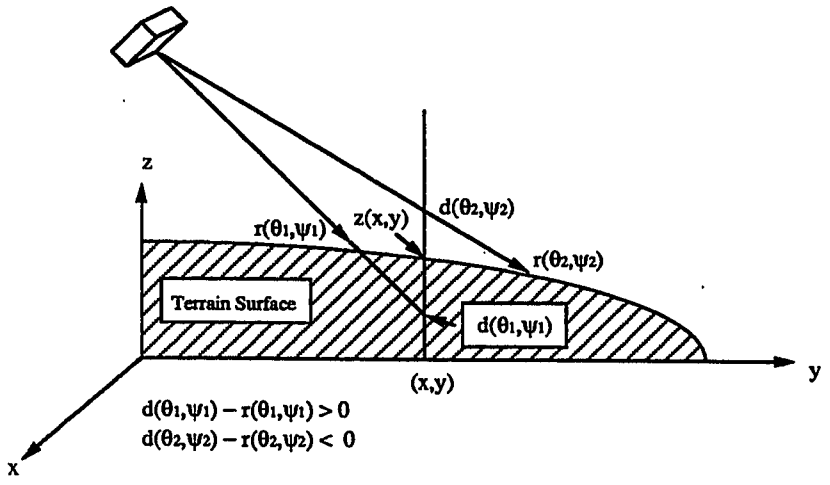


Figure 4-8. Scan lines between which the vertical line intersects with the terrain surface.

Let

$$\Delta\psi_i = d_i(\psi_i) - r_i(\theta_i'(\psi_i),\psi_i)$$

where

$\theta'(\psi)$ is the closest column point to $\theta(\psi)$ and

r is the range data at $(\theta'(\psi),\psi)$.

The two scan lines are located such that $\text{sign}(\Delta\psi_1) \neq \text{sign}(\Delta\psi_2)$. Second, a binary search between the scan lines is performed until $|\psi_n - \psi_{n+1}| \leq \varepsilon$ as shown in Figure 4-9. Since there are no data between the scan lines, Lagrangian interpolation uses the four range data surrounding the intersecting point as control points to find a value ψ, where $\psi_1 < \psi < \psi_2$. ψ is then processed by the Cartesian conversion to

find the elevation for the missing point. Repeating this algorithm for every missing point can produce a dense elevation map of desired resolution.

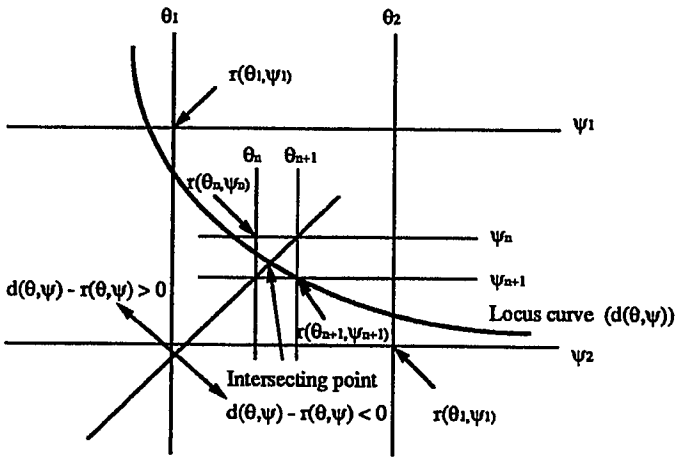


Figure 4-9. Lagrangian Interpolation using the adjacent range points as control points to find the intersecting point of the curve with the terrain surface.

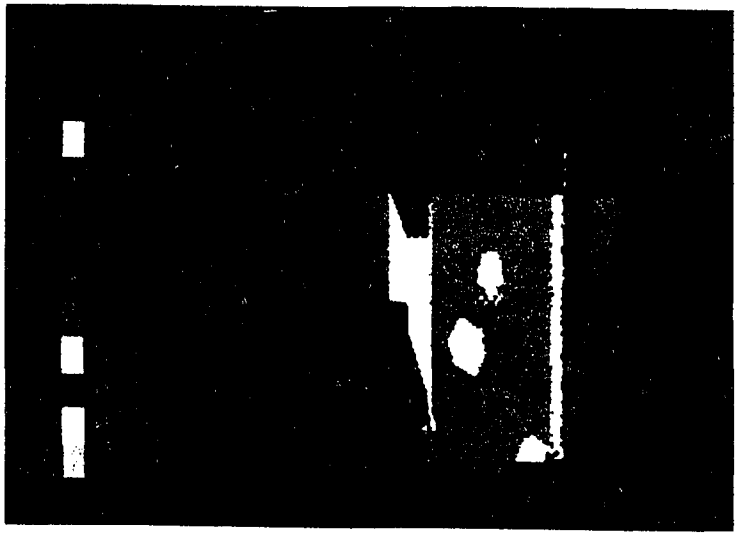


Figure 4-10. Terrain map generated by the locus algorithm.

Experiments (see Figure 4-10) demonstrated that this algorithm is very powerful in interpolating range data but it is extremely time consuming (due to large number of iterations). The locus algorithm averaged 31.41 msec to find the elevation of a missing data compared to 0.86 msec per point for direct Cartesian conversion (tested on a 20-Mhz Compaq-386 personal computer).

4.7 Range Shadows

Because of the geometry of the scanner, shadow or occluded regions on the back side of obstacles are not visible to the laser (see Figure 4-11). If the locus algorithm is applied to the missing point inside these regions, the surface may be interpolated incorrectly. Missing points inside the shadow area will have to be distinguished from known areas. It would be improper to interpolate across the shadow area unless there was some reliable knowledge of the environment.

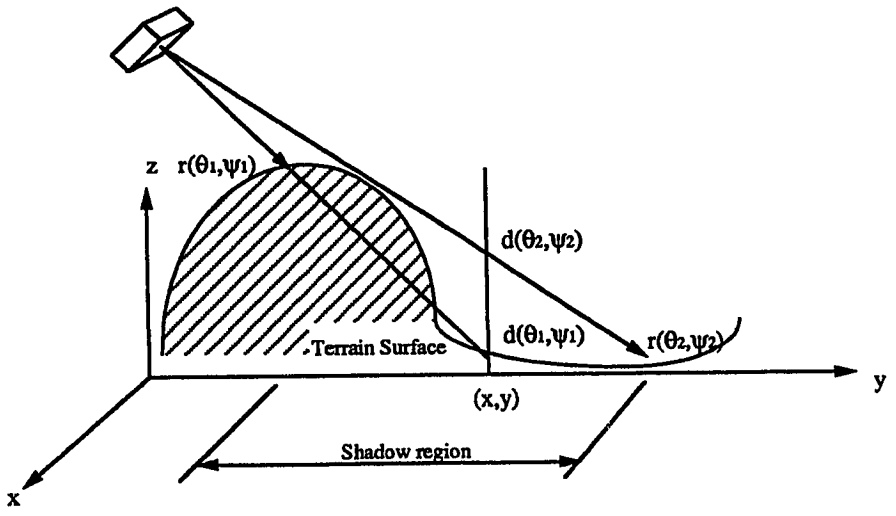


Figure 4-11. Shadow region behind an object.

The locus algorithm can also be used to detect shadow regions. An (x,y) location in the map is in a shadow area if its locus intersects the image at a edge point. This is established by observing that a range shadow correspond to an occluding edge in the range image. As shown in Figure 4-11, there exists a discontinuity between the range measurement of the scan line that strikes the top of the object and its adjacent scan line that strikes the background behind the object. This type of discontinuity is called a *jump edge* and signals the existence of shadow area. When the locus algorithm is applied at a jump edge, the location is marked as a shadow point.

4.8 Edge Detection

Edge detection is a fundamental low-level process in a scene analysis system. There are two types of edges of interest: jump edges and roof edges. The jump edge is defined as discontinuities of the depth value in the range data. Such edges occur when an object occludes another object or when part of an object occludes itself. Roof edges correspond to surface creases; points over which surface normals are discontinuous. Many techniques have been developed for extracting edges from range data [Inokuchi 82, Mitiche 83, Svetkoff 84, Bhanu 86, Daily 87]. A good survey paper on edge detection is given by Besl and Jain [Besl 85]. Among those edge detectors, only the one developed by Mitiche and Aggarwal [Mitiche 83] accommodates range noise. However, Mitiche's algorithm only detects a roof edge with an assumed orientation. Our research developed a noise-insensitive edge detector. It can be used whenever range data contains considerable amount of range

noise.

From its definition, a jump edge can easily be detected by searching for a discontinuity or jump between two adjacent range points that exceed a threshold value. This operation is analogous to the gradient edge detector for intensity images. With a laser system, however, smaller range differences are expected between values that are near the laser than those further away. Because of this, the threshold value must be a function of distance rather than a constant value. That is, a jump edge is detected if

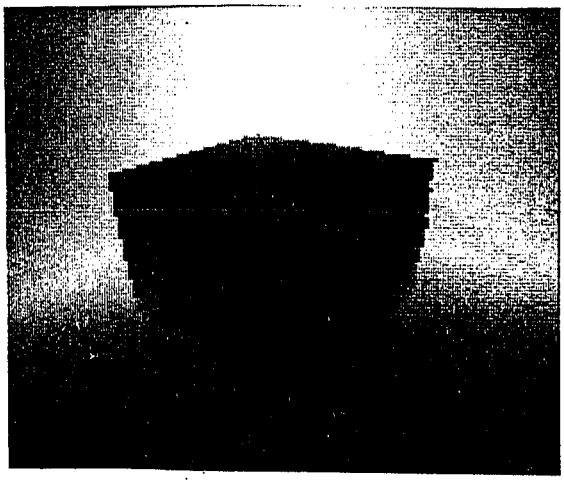
$$|r(\theta_1,\psi_1) - r(\theta_2,\psi_2)| \geq T_{threshold}\, r(\theta_1,\psi_1) \qquad (4.3)$$

A threshold of .06 ($T_{threshold}$ = .06) is currently being used in our research. This allows for up to 6% difference between range values without being declared an edge point. As shown in Figure 4-12, this edge operator works fairly well for a noise-free range image. With noise present, however, the threshold should be increased to accommodate the noise. Nevertheless, the number of false edges will also increase as the threshold increase. Figure 4-13 shows the edges generated from noisy range data.

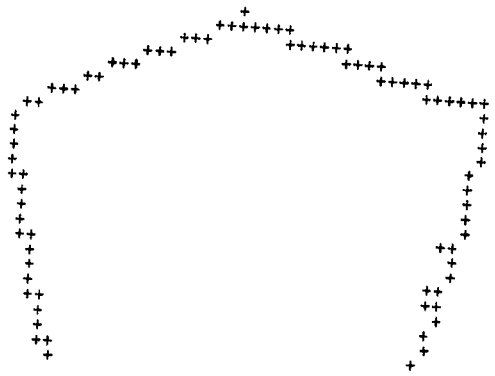
4.9 A Noise Insensitive Edge Detector

Edge detection is the problem of determining whether an edge (jump or roof edge) is present at a point p. Let N_i, i=1,2,3,...,N, denotes the N mutually exclusive regions sampled from the points adjacent to p as shown in Figure 4-14. Each

region is known by M samples taken from the region. Let's assume temporarily that the sample points for each region are *properly sampled.* This means that the sample points representing a particular region i must lie on the same surface. The edge detection problem can then be formulated as a binary hypothesis test, in which

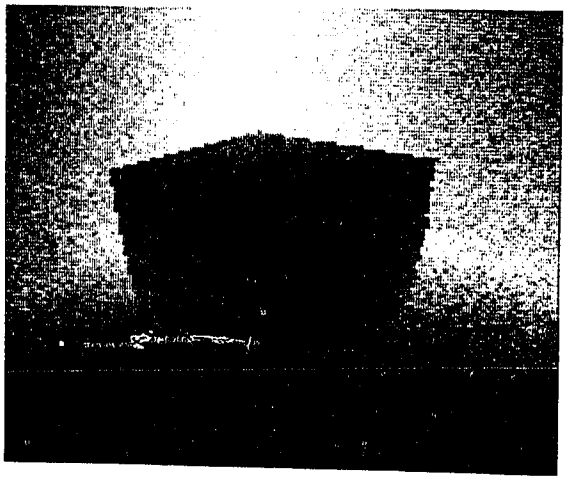


(a)

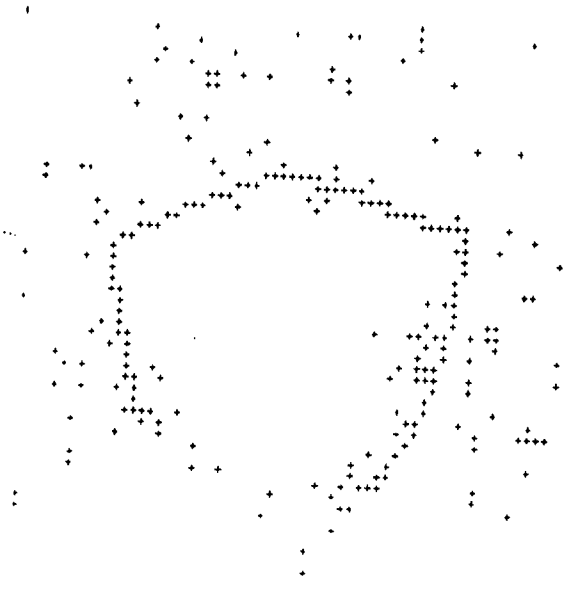


(b)

Figure 4-12. (a) Laser image of a cubic box sitting at a corner of a room, (b) jump edges output form Equation 4.3.



(a)



(b)

Figure 4-13. (a) A normally distributed random noise is added to the range image of the cubic box, (b) jump edges output from Equation 4.3.

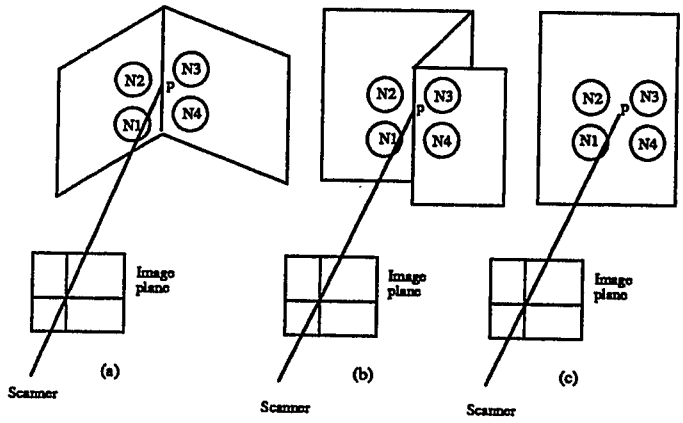


Figure 4-14. Mutually exclusive sample regions around (a) roof edge (b) jump edge, (c) plane.

H_0: edge is present at p if N_i's are sampled from at least two different surfaces as shown in Figure 4-14a and 4-14b.

H_1: edge is not present at p if N_i's are sampled from the same surface as shown in Figure 4-14c.

The decision regarding the hypothesis is made by performing the following steps.

(1) Planar surfaces are fitted to each region using the sampled data. Details of fitting a planar surface to a set of 3-D points are given in Appendix B. Let $\mathbf{n}_i$ denote the surface normal of the best fitted surface and μ_{ri} denote the averaged distance from the scanner to the sampled data for region i.

(2) The maximum angle θ_{max} between the calculated surface normals and the maximum difference d_{max} between these averaged distances are computed. That is,

$$d_i = \frac{1}{M} \sum_{j=1}^{M} r_{ij},$$

$$\theta_{max} = \text{Max}[\cos^{-1}(\mathbf{n}_i \cdot \mathbf{n}_j),\ i=1,2,...,N,\ j=1,2,...,N,\ i \neq j]$$

$$d_{max} = \text{Max}\left[\frac{|d_i - d_j|}{d_i},\ i=1,2,...,N,\ j=1,2,...,N,\ i \neq j\right]$$

where

r_{ij} is the range value of the sample point j in region i.

$(\mathbf{n}_i \cdot \mathbf{n}_j)$ denotes the inner product of $\mathbf{n}_i$ and $\mathbf{n}_j$.

(3) Once θ_{max} and d_{max} have been determined, the hypothesis test can be asserted by

H_0 is true if

1. $\theta_{max} >= \theta_{threshold}$, a roof edge is present at *p*,
2. $d_{max} >= d_{threshold}$, a jump edge is present at *p*.

Otherwise,

H_1 is true.

θ_{max} and d_{max} can be determined using the a priori knowledge of the objects in the scene. But how does one know the points are properly sampled? If the points are sampled from two different surfaces as shown in Figure 4-15, false edge point may be declared. Let x_{ij} denote the Cartesian coordinates of the range data j in region i and the minimum square error be the cost criterion for surface fitting; the error *E* returned from the surface fitting routine is then given by

$$E_i = \sum_{j=1}^{M} e_{ij}^2 = \text{Min}[\mathbf{n}_i^t \, \mathbf{A}_i \, \mathbf{n}_i]$$

where

$\mathbf{n}_i$ is the surface normal of the best fitted plane,

$$\mathbf{A}_i = \sum_{j=1}^{M} \mathbf{x}_{ij} \, \mathbf{x}_{ij}^t,$$

$\mathbf{n}^t$ denotes a transposed vector.

If the characteristics of the noise are known, a proper threshold value can be chosen accordingly for determining whether the points belong to same surface. That is,

if $E_i > E_{threshold}$,

then it is declared that the sample points do not lie on the same surface and the fitted surface is discarded. There are four possibilities when making such a decision:

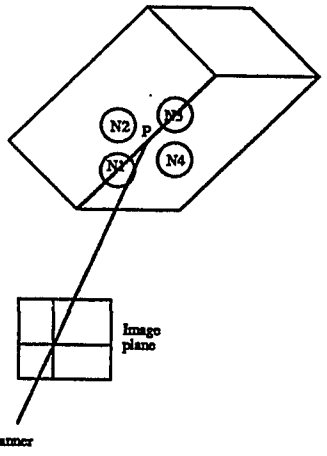


Figure 4-15. N_1 and N_3 contains sample points from two different surfaces. Only N_2 and N_4 will be used for making edge decision.

1) Sample points are properly sampled but

$E_i > E_{threshold}$ (type I error),

2) Sample points are properly sampled and

$E_i <= E_{threshold}$

3) Sample points are not properly sampled and

$E_i > E_{threshold}$

4) Sample points are not properly sampled but

$E_i <= E_{threshold}$, (type II error).

As discussed in section 3.10, the range measurement error of an AM laser scanner is known as zero mean normally distributed random noise. Let $p_{e/\sigma}(E|\sigma,y)$ denote the probability density function of the minimum squared error given the standard deviation of the random noise and the fact that all points lie on the same surface. Also let $p_{e/\sigma}(E|\sigma,n)$ denote the probability density function where the sample points do not lie on the same surface. The goal is to find a threshold value that minimizes both types of error (type I and type II errors). As shown in Figure 4-16, if $p_{e/\sigma}(E|\sigma,y)$ and $p_{e/\sigma}(E|\sigma,n)$ are both known, $E_{threshold}$ can be determined by the intersecting point of the density functions. Unfortunately, it is impossible to model $p_{e/\sigma}(E|\sigma,n)$ since many other possible situations exist where the points are not sampled from the same surface. For examples, the points can be sampled from two adjacent surfaces sharing a roof edge or two surfaces far apart as depicted in Figure 4-17. However, it can be proven, in Appendix B, that $p_{e/\sigma}(E|\sigma,y)$ is a Ki-square distribution function with M-3 degree of freedom (M is the number of sample points). After knowing $p_{e/\sigma}(E|\sigma,y)$, one can choose a proper threshold value

corresponding to the allowed Type I error. For example, if 10 sample points are used, the least squared error obeys the Ki-square distribution with degree of freedom 7 as depicted in Figure 4-18. From this figure, it is found that there is a 92% chance that the least squared error is smaller than 13. In other words, if $E_{threshold}$ is set to 13, the probability of a type I error occurring is 8%.

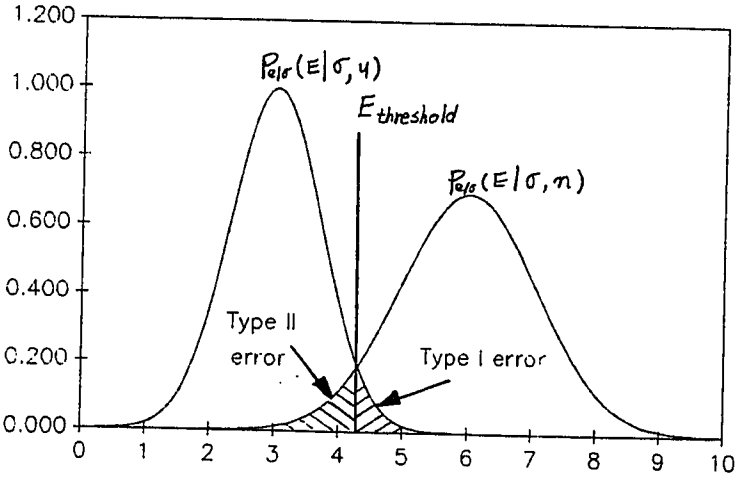


Figure 4-16. Type I and Type II error determined by the threshold value.

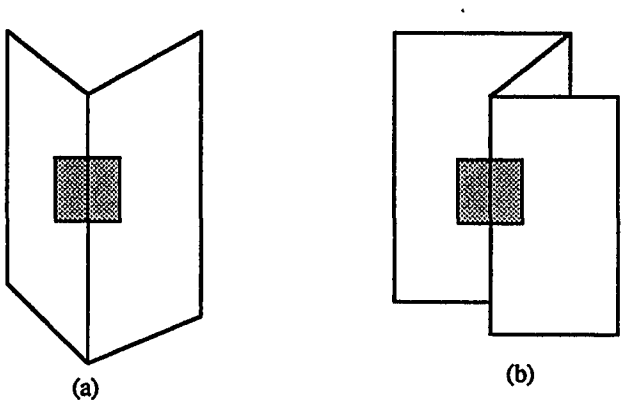


Figure 4-17. Points sampled from (a) adjacent surfaces, (b) surfaces far apart.

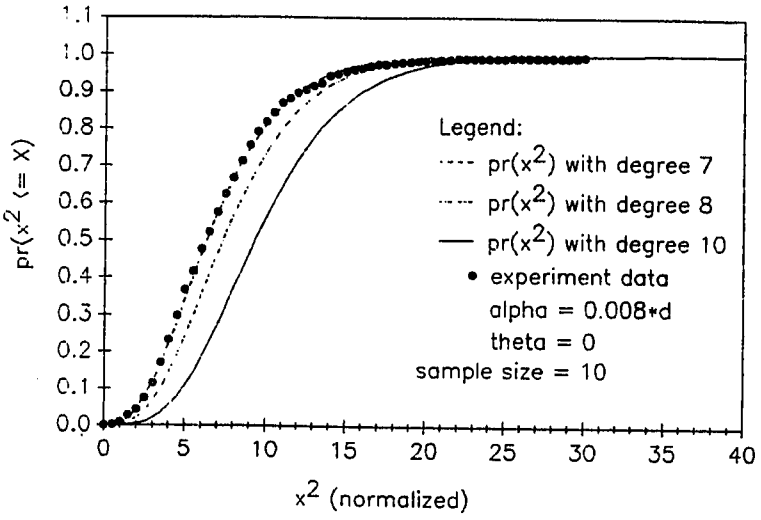


Figure 4-18 $p_{\epsilon/\sigma}(E \mid \sigma, y)$ obeys the Ki-square distribution.

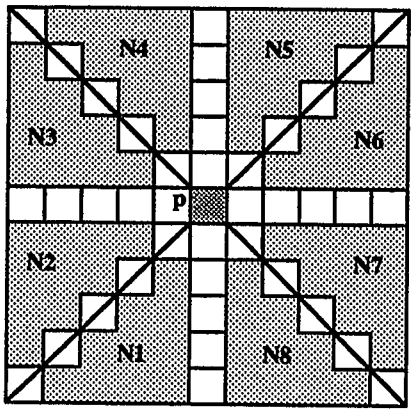


Figure 4-19. An 11x11 neighborhood centered at the candidate point is used for edge detection.

In our experiments, edges are detected using an 11 x 11 neighborhood as shown in Figure 4-19. Eight regions, each containing 10 points, are obtained from the neighborhood. Noisy data, as shown in Figure 4-13(a), were used to test this

algorithm. A zero mean normally distributed random noise was added to the range data. The standard deviation of the random noise was 1.2% of the range value. This represents an appreciable amount of noise since approximately 5% of the errors are greater than 12 cm and 32% of the errors are greater than 6 cm at a distance of 5 meters. This noise is considerably higher than those allowed by commercial laser imaging systems [Odetics 87, Beyer 87]. The computed edges using the edge detector described here are shown in Figure 4-20. Figure 4-21 shows the final edge output by extracting the center edge point along the direction perpendicular to the edge and removing minor isolated edge points.

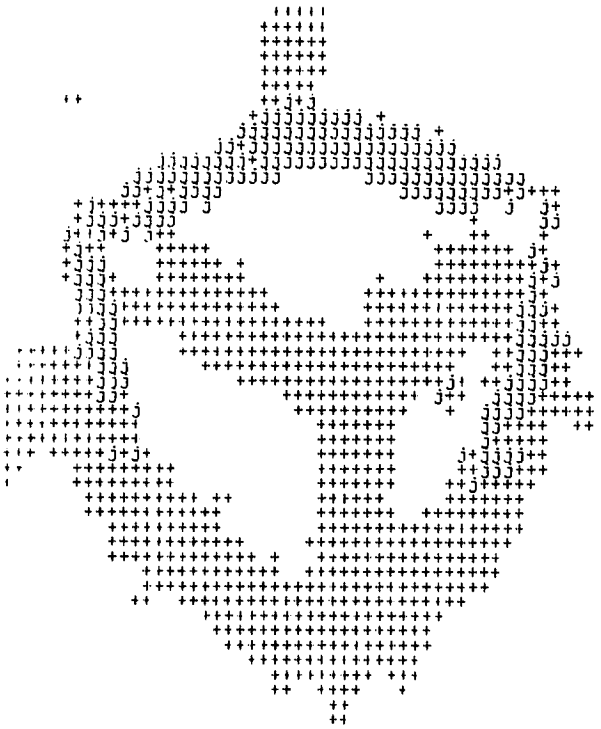


Figure 4-20. Edge map of the noisy cubic scene in Figure 4-13(a) generated from the noise-insensitive edge detector (j's represent jump edges and +'s represent roof edges)

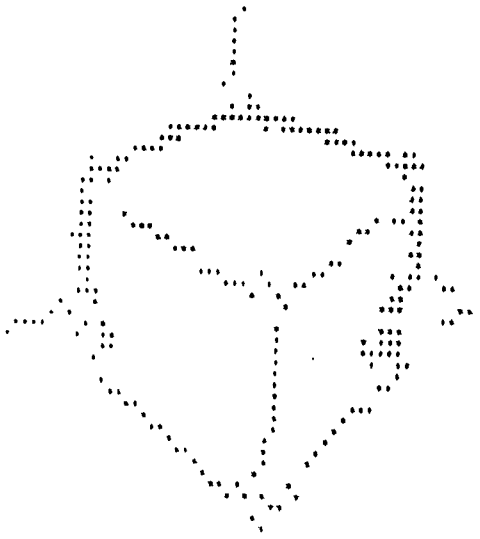


Figure 4-21. Final edge output.

4.10 Detection of Erroneous Range Measurements Due to Weak Returned Signal

Besides the missing points discussed in the preceding sections, there is another type of range data of interest: the range data with small or zero reflectance values. Since the reflectance data are proportional to the power of the returned laser, small reflectance values mean a weak returned laser signal. Recall the noise model developed by Nitzan *et. al.*,

$$\sigma_r = \frac{1}{2\sqrt{2}\,\pi}\,\frac{\lambda_m}{m\ \mathrm{SNR}}$$

$$\mathrm{SNR} = \left(\frac{\alpha\eta\lambda A_R F_T T}{\pi hc}\,\frac{\rho_d \cos(\theta)}{r^2}\right)^{1/2}$$

When the received signal is weak ($\rho_d \cos(\theta)/r^2$ is small), the range measurement noise σ_r becomes significant. This means that the range value measured from a weak returned signal is actually a normally distributed random value and should not be used for any further processing.

To understand why this happened, let's review how noise is generated in a real laser scanner. Noise in the laser ranging device mainly arises during the detection process. There are two major types of noise: dark current shot noise and photon fluctuation noise [Pratt 69]. The shot noise is caused by dark current flowing in the photodetector in the absence of external photoexcitation. The photon fluctuation noise is caused by emission fluctuation of the optical radiation incident upon the photodetector. Therefore, the ranging accuracy depends on the power and stability of the received signal. For laser scanners employing an amplitude modulated CW laser, photon fluctuation noise is inevitable. Once the modulation frequency is determined, however, this noise can be alleviated with a noise reduction filter. The shot noise is the major source of noise. In the absence of a returned laser signal, shot noise dominates the output signal and causes erroneous range measurements. Other noise sources also degrade the range measurements. Examples include transmission noise, ambient noise, and amplifier noise. However, these types of noise are relatively insignificant compared with the shot noise [Nitzan 77].

Four kinds of targets that produce weak returned signals: sky or wide open space, objects bounded by mirror-like surfaces, objects viewed obliquely, and dark distant objects. Since these objects are practically *invisible* to the laser scanner,

range points measured from these objects will be labeled as invisible points.

Using reflectance data, erroneous range data can be detected easily. The next task is to correct them. However, whether these erroneous range points will cause problems and require corrections depends ultimately on the application. For autonomous navigation, the goal is to convert the range data into an elevation map. During the conversion process, range data with small reflectance values will simply be discarded. No correction will be made. This will result in additional missing points on the elevation map. This type of missing point can again be detected with the locus algorithm. A missing point whose locus intersects the image at invisible points will be labeled as an invisible point. The elevation map produced from this operation will then contain invisible regions. Nevertheless, these regions provide an important piece of information: there are objects that are not *visible* to the laser scanner but that *exist* at the locations represented by these regions. Based on this observation, it is safe to simply treat these regions as areas occupied by obstacles. For other applications in which object recognition is required, other devices such as tactile sensors [Allen 87] can be used for extracting geometric information about the objects.

4.11 Conclusion

Procedures for developing topographical terrain maps from range data are presented in this chapter. The terrain maps generated in this research consist of a gridwork of 128 by 128 points. Each grid cell represented a 1.125"x1.125" square area. Grid points in the terrain map are classified into four types: unscanned,

scanned, shadow, and invisible. Unscanned points represent the area outside the field of view of the laser scanner and the scanned points contain elevation values of the terrain surface. Occluded areas are represented by shadow points and the area occupied by *invisible* objects will be identified by invisible points. A natural terrain path planner which utilizes these topographical terrain maps to pilot a mobile robot in a dynamic environment was developed in this research. This path planner will be discussed in the next chapter.

Chapter 5

Sensor-based Direct Search Algorithm (SDSA)

for

Autonomous Natural Terrain Navigation

5.1 Introduction

Navigation involves planning an acceptable route from a starting point to a destination. Autonomous navigation means to use an intelligent machine without human assistance to accomplish this task. Much research effort in the past has concentrated on the problem of navigation in an indoor (2-D) environment and numerous techniques now exist for 2-D path planning. Examples include the well-known Lozano-perez A* graph searching algorithm and its descendants [Iyengar 85, Monaghan 85, Javis 88, Arkin 88, Zaharakis 86] and the potential algorithms [Khatib 86, Hwang 88]. Using a configuration space (CS) model (as discussed in section 4.2), the graph searching algorithm generates paths by connecting line segments between vertices unobstructed by obstacles. A minimum distance path can be found by using the length of the line segments as the cost criterion. The graph searching algorithm is a powerful tool for finding optimal paths in a static environment (where the CS model is given). For navigation in a dynamic environment, however, this algorithm has the disadvantage of requiring extensive computations to convert sensory data to CS models.

The potential field (PF) algorithm was proposed originally by Khatib [Khatib 86] as a solution for controlling a robot manipulator working in a crowded work

space. Hwang and Ahuja [Hwang 88] later used this algorithm for planning collision-free paths for the *moving-piano* problem. The PF algorithm represents the navigation space as a binary map indicating obstacles and free spaces. An artificial potential representing a repulsive force is assigned to each obstacle. The repulsive force prevents the *moving-piano* from colliding into obstacles while moving toward the goal. The advantage of the potential field is that the binary map can be easily obtained from sensory data. However, the drawback to this algorithm is that it often generates irrational paths due to repulsive movements caused by obstacles.

There are other navigation algorithms, (the Two-Sweep algorithm developed by Gilmore *et. al.* [Gilmore 84] and the Heuristic algorithm by Chattergy [Chattergy 85]), that work fairly well for 2-D navigation. But the real problem is devising algorithms for natural terrain (3-D) environments.

5.2 Natural Terrain navigation

What are the difficulties in navigation in a natural terrain environment? The major one is obstacle detection. *Obstacle* in 3-D navigation space is not yet clearly defined. The most widely accepted definition [Daily 87] defines obstacles as *regions that are nontraversable.* Still, this definition is questionable. Nontraversable regions can become traversable when a different means of mobility is used. For example, a small object may be considered an obstacle to a wheeled robot but can easily be walked over by a legged robot. Few natural terrain navigation planners have been developed because a systematic procedure for extracting *obstacles* from 3-D sensory data has not been available.

Quek *et. al.* [Quek 85] proposed a rough terrain navigation algorithm similar to the PF algorithm that represents the navigation space as a 2-D cellular array. The value in each cell is the traversal cost (a multi-valued quantity, rather than a binary-valued one used by the PF navigation planner) of that cell. By using the cost as the refractive index of that cell, a propagating wave centered at the starting point was generated. The wave propagates rapidly over cells of low refractive index (cost) and more slowly in regions of high refractive index. Then a minimum cost path can be generated by tracing the normals of the wave front from the goal to the starting point. This algorithm has the disadvantage of requiring extensive computations to generate the cost map, propagated wave, and tracing algorithm. Moreover, the constant cost assigned to each cell is impractical. The traversal cost of a cell should be a function of instantaneous speed and the robot's heading at that point rather than a constant value. Also, the irregular shape of the propagated wave causes the algorithm to generate irrational paths.

Gaw and Meystel [Gaw 86] developed a 3-D version of the graph searching algorithm in which the navigation space is modeled by 3-D polygons. Graph searching is performed using the 3-D vertices of these polygons. Like its 2-D precedent, this algorithm is practical only in a static environment.

Daily *et. al.* [Daily 87, 88] at Hughes A.I. Center developed a natural terrain navigation system that generates collision-free paths using segmented elevation maps. Obstacle regions are detected by fitting surface patches to the range data and using a model of the vehicle to test each surface patch for traversability. A simple but practical navigation strategy uses straight line segments in the traversable regions to

obtain the navigation path. This system provides plausible performance on natural terrain. Field-tests [Daily 88] have demonstrated its ability of autonomous off-road navigation with obstacle avoidance. However, this system does not provide smooth motion of the vehicle on the planned paths; since the paths consist of straight line segments and instantaneous changes in direction are physically impossible. Errors in position and heading of the mobile vehicles at the discontinuous points on the path are thus inevitable. In addition, optimality of the planned path is not a consideration in the navigation scheme. To obtain a smooth motion, Kant and Zucker [Kant 87] proposed a post-processing technique for smoothing the line segments and the output path is also guaranteed to be collision-free. Nevertheless, the smoothed path contains B-spline segments rather than a global parametric equation that describes the entire path.

In the terrain where smooth paths are available for navigation, it is desirable to extract smooth paths directly from the sensor data. To achieve this goal, this thesis develops a 3-D sensor-based direct search algorithm (SDSA). Using elevation maps produced by the perception system described in the preceding chapter, SDSA generates collision-free optimal paths of any desired degree of smoothness readily usable by the steering mechanism of mobile robots. Gradient and elevation tests check the traversability of every candidate path instead of detecting obstacles over the entire range data.

5.3 Mathematical Formulation

As shown in Figure 5-1, natural terrain navigation can be addressed as a problem of finding an optimal path

$$R(t) = x(t)\mathbf{i} + y(t)\mathbf{j} + z(t)\mathbf{k},\ t \in [0,1], \tag{5.1}$$

on a given surface,

$$z(t) = f(x(t),y(t)), \tag{5.2}$$

subject to the boundary condition,

$$R(0) = x(0)\mathbf{i} + y(0)\mathbf{j} + z(0)\mathbf{k} = p_0 = x_0\mathbf{i} + y_0\mathbf{j} + z_0\mathbf{k}, \text{ and}$$
$$R(1) = x(1)\mathbf{i} + y(1)\mathbf{j} + z(1)\mathbf{k} = p_1 = x_1\mathbf{i} + y_1\mathbf{j} + z_1\mathbf{k}.$$

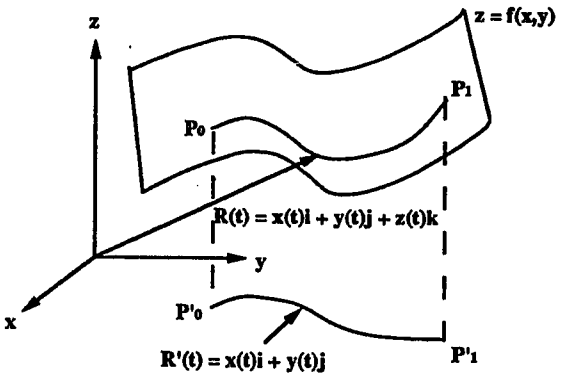


Figure 5-1. Natural terrain path planning.

The term "optimal path" is used because the chosen path optimizes the cost criterion. Examples include the minimum distance and minimum energy criteria. Let the parametric polynomials

$$x(t) = \sum_{i=0}^{n} a_i\, t^i$$

$$y(t) = \sum_{i=0}^{m} b_i\, t^i, \tag{5.3}$$

describe the projection of the path on the x-y plane. Since the path is constrained to the terrain surface for land navigation, $z(t)$ is completely determined by $x(t)$ and $y(t)$ and the terrain surface. Natural terrain path planning is thus equivalent to a problem of finding the projected path on the x-y plane. Substituting the boundary conditions into Equation 5.2 produces

$$a_0 = x_0, \quad b_0 = y_0,$$

$$a_1 = x_1 - x_0 - \sum_{i=2}^{n} a_i, \text{ and} \qquad b_1 = y_1 - y_0 - \sum_{j=2}^{m} b_j \tag{5.4}$$

After combining the above equations, the path $\boldsymbol{R(t)}$ can be expressed as a function of n+m-2 parameters. That is

$$\boldsymbol{R(t)} = f(a_2, a_3, \ldots, a_n, b_2, b_3, \ldots, b_m)^t$$

Let

$$C = \int_0^1 g(\boldsymbol{R(t)})\, dt \tag{5.5}$$

be the cost function to be optimized, where g is a scalar function. Path planning can be expressed as an unconstrained optimization problem of finding parameters a_i's and b_j's (i = 2,3,...,n and j=2,3,...,m) that minimizes the cost function. That is,

Minimize $\quad C = \int_0^1 g(\boldsymbol{R(t)})\, dt$

where

$\boldsymbol{R(t)} \in \mathbf{E}^{n+m-2}$,

$\mathbf{E}^{n+m-2}$ denotes a n+m-2 dimensional parameter space.

Each point in the parameter space represents a path on the given terrain. Figure 5-2 illustrates the mapping between points in the parameter space and the navigation paths for $R(t) \in E^2$. Knowing the cost of each point (path) in the parameter space, one can obtain the optimal path by using existing optimization procedures to find the optimal point.

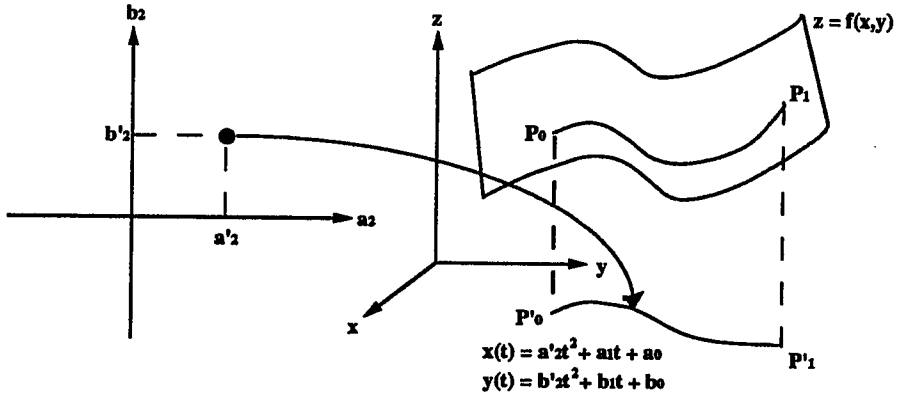


Figure 5-2. Mapping between points in the parameter space and navigation paths.

When obstacles are present, a traversability test will be performed on each candidate path during the searching process. Penalties will be added to the cost of every path failing the traversability test. That is,

$$C' = \int_0^1 g(R(t))\, dt + k \int_0^1 P(R(t))\, dt \qquad (5.6)$$

where

k is the penalty constant ($k >> 0$),

$\int_0^1 P(R(t))\,dt$ is the traversability test (or penalty function)

To insure convergence [Luenberger 84], the penalty function must satisfy the following conditions:

(i) $\int P(R(t))\,dt$ is a continuous monotonic function,

(ii) $\int P(R(t))\,dt >= 0$ for all $R(t) \in E^n$,

(iii) $\int P(R(t))\,dt = 0$ if and only if $R(t)$ passes the traversability test.

The traversability test is performed by moving a vehicle model along the path to check its traversability. A simple penalty function can be the length of the path failing the traversability test.

5.4 Direct Search Algorithm

There are two types of optimization procedures available for solving unconstrained minimization problems: the direct search method and the gradient method. Examples of the direct search method include the direct searching algorithm developed by Hooke and Teeves [Hooke 60] and the Nelder-Mead simplex searching algorithm [Nelder 65, Olsson 74]. Newton's method and the conjugate gradient method [Luenberger 84] are examples of the gradient method. In general, gradient methods converge to a solution much more rapidly than direct search methods if the cost function and its gradient can be evaluated at every point

in the parameter space. For autonomous navigation where the terrain is known only by its samples (the discrete elevation maps), gradient of the cost function cannot be evaluated unless one can obtain dense points along the path. For this reason, the optimization process employed in our research used the Nelder-Mead simplex searching algorithm which only requires evaluation of a cost function.

The simplex searching algorithm guarantees convergence to a solution if it exists. For a problem involving n parameters, a *simplex* of n+1 vertices is used. The cost associated with each vertex of the simplex is calculated and then a sequence of flipping and contractions of the simplex along the direction of minimum cost are performed. Figure 5-3 shows an example of simplex searching on a 2-D (n=2) cost map. Details of this algorithm can be found in the paper by Olsson [Olsson 74]. Because the path planner developed in this thesis uses a direct search optimization procedure and evaluates the cost function using an elevation map derived from the sensory data, it was named the Sensor-based Direct Search Algorithm (SDSA).

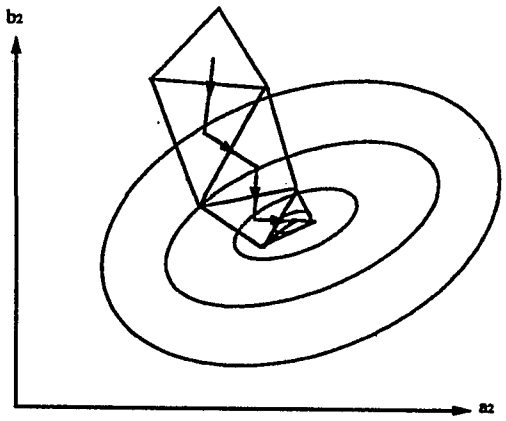


Figure 5-3. Simplex searching in a 2-D map.

5.5 Minimum Distance Collision-Free Path Planning

To illustrate the SDSA algorithm, let's consider a simple case of finding minimum distance collision-free paths using elevation maps. The length of a path is given by

$$l = \int_0^1 \left| \frac{dR}{dt} \right| \, dt$$

$$= \int_0^1 (x_t^2 + y_t^2 + z_t^2)^{1/2} \, dt$$

where

$x_t = dx / dt$, $y_t = dy / dt$, $z_t = dz / dt$.

Given the parametric equations $x(t)$, $y(t)$ and the discrete elevation map $z(x(t),y(t))$, the length can be approximated by

$$l = \sum_{i=1}^{N} ([x(i\Delta t)-x((i-1)\Delta t)]^2 + [y(i\Delta t)-y((i-1)\Delta t)]^2 + [z(i\Delta t)-z((i-1)\Delta t)]^2)^{1/2} \quad (5.7)$$

where

$z(i\Delta t) = z(x(i\Delta t), y(i\Delta t))$, is the elevation of a step point on the path.

$\Delta t = 1/N$ is the step size for evaluating the length.

Since there is no one-to-one relation between the step points along the path and the grid points in the elevation maps as shown in Figure 5-4, linear interpolation will be used to evaluate $z(i\Delta t)$ using the four adjacent grid points.

A traversability test which returns the length of the path intersecting with the forbidden regions is used as the penalty function. The term *forbidden regions* is used here to include the shadow and unscanned areas which may be traversable but are forbidden because of lack of information. Using the length as the cost of the path, the problem of finding a collision-free minimum distance path is transformed into an optimization problem. That is

$$\text{Minimize} \qquad C' = l + k \int_0^1 P(R(t))\, dt \qquad (5.8)$$

where

$R(t) \in \mathbf{E}^{n+m-2}$.

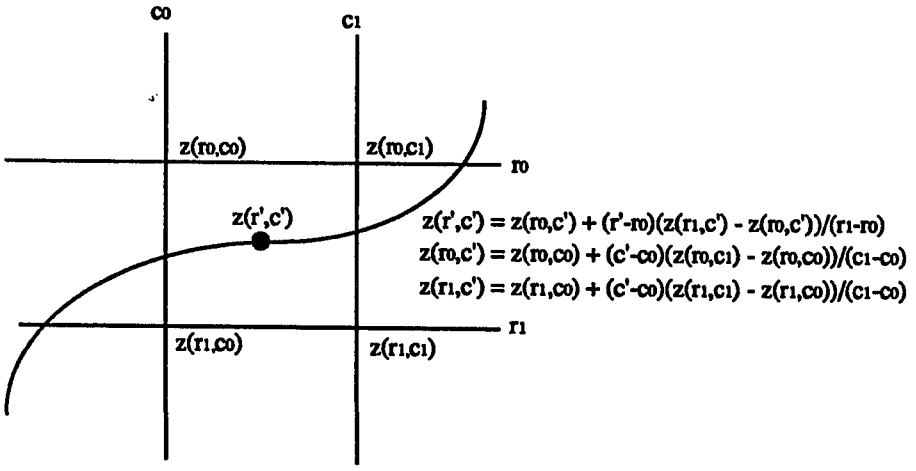


Figure 5-4. Linear interpolation to find the elevation of a point on the path.

5.6 Experimental Results

The artificial robot world described in section 3.13 was used to test the SDSA algorithm using minimum distance as the optimization criterion. Figure 3-20a shows the range data (generated by LISA) of the scenes in the artificial robot world. The elevation maps corresponding to these range data are shown in Figure 4-10. Minimum distance paths generated by the SDSA algorithm are shown in Figure 5-5. Notice the double tracks of the navigation path in Figure 5-5. The width of the track represents the width of the mobile robot plus a safety tolerance. The traversability test was performed not only along the center line of the path for elevation clearance check between the robot platform and the ground surface, but also along both of the tracks for traversability check. Figure 5-6 shows an artificial elevation map of a low hill surrounded by obstacles resembling small high hills and Figure 5-7 shows the minimum distance paths generated by SDSA algorithm on this terrain.

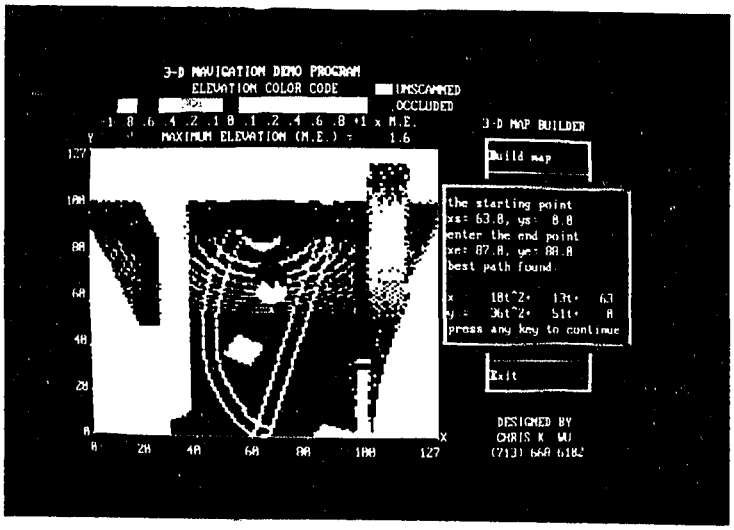


Figure 5-5. Minimum distance paths generated by the SDSA algorithm.

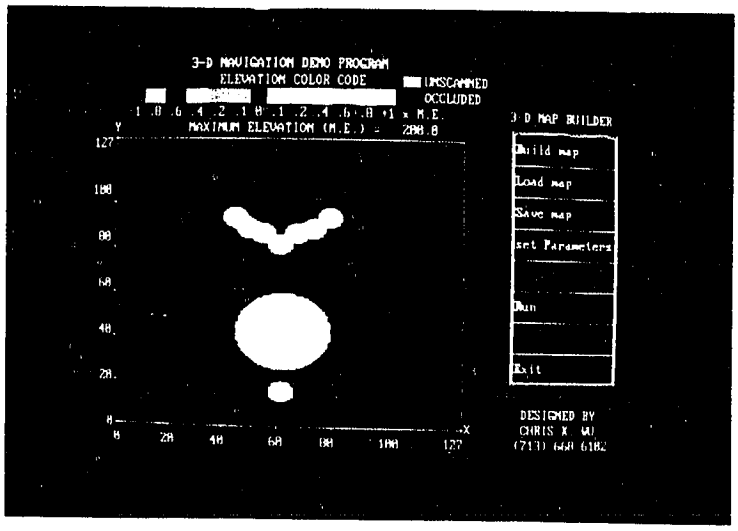


Figure 5-6. An artificial elevation map.

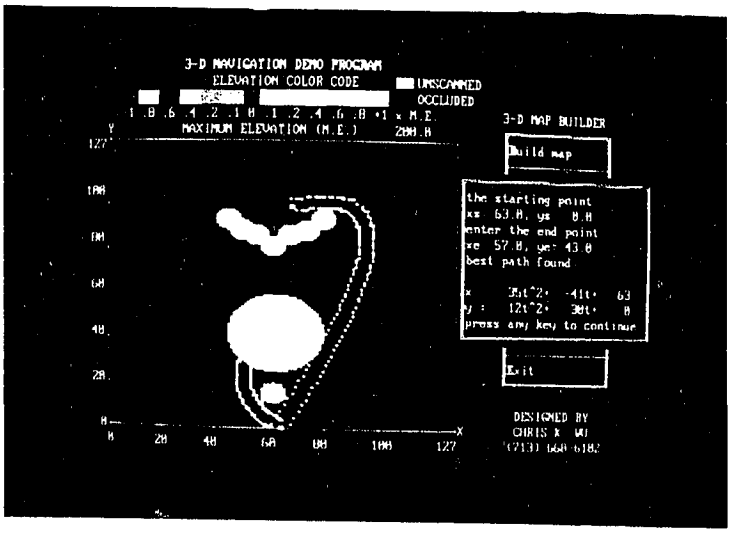


Figure 5-7. Minimum distance paths generated by the SDSA algorithm.

5.7 Collision-Free Path Planning for Multiple Mobile Robots

Once the optimal path has been planned, the next task is to determine the velocity of the mobile robot along the path. This is the problem of *mobility planning*. Mobility planning is less significant if the mobile robot is the only moving object in its environment. However, it becomes important when multiple robots work in the same environment. The mobility planning problem is best explained with the following example. In Figure 5-8 consider the robot that travels from point $R(0)$ at time s_0 to point $R(1)$ at time s_1. Let s denote the time variable to distinguish from the spatial variable t used for describing the path. The straight line between the two points represent the optimal path generated by the path planner. Considering a second robot whose path intersects with the planned path of the first one. It is clear that different velocities along the path will provide either a collision-free path or one that collides into moving robots.

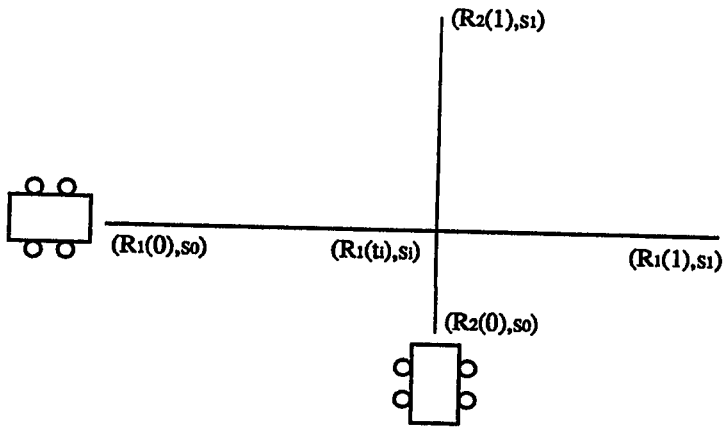


Figure 5-8. Mobility planning

To illustrate how the SDSA algorithm is used to solve the mobility planning task, let's first introduce the spatial-time (s-t) map. In Figure 5-9 the s-t map is a 2-D map in which the abscissa is the time variable *s* and the ordinate is the spatial variable *t* describing the position along the planned path. The origin of the s-t map represents the starting point of the robot and the (1,1) point represents the destination of the robot. The velocity profile of the mobile robot can be described by the curve connecting these two points in the s-t map. A straight line, for example, represents a nearly constant velocity along the path. If the velocity must be zero at the starting and destination points and steady between these two points, smooth curves with zero slope at both end points will be used to describe the velocity as shown in Figure 5-9.

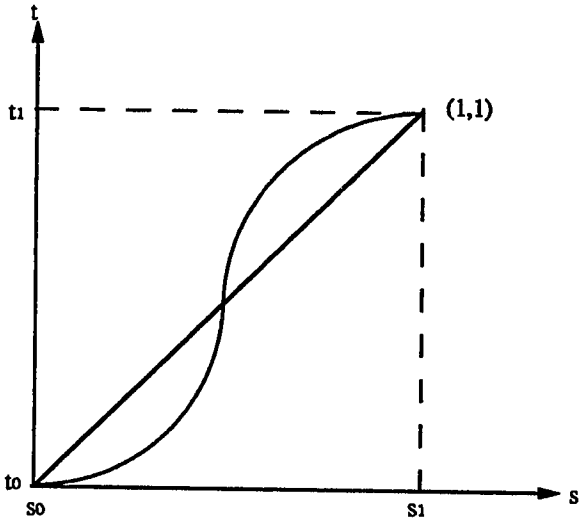


Figure 5-9. Velocity profile described by the s-t curves.

Using the two-robot example descried in the previous paragraph, the velocity curves of the moving robots are obtained as follows:

(1) Compute the velocity of the first robot without considering the

presence of the second robot. This generates a velocity curve in the s-t map based on the task and hardware specifications of the mobile robot.

(2) Calculate the time period $[s_i,s'_i]$ and the spatial interval $[t_i,t'_i]$ between which the first robot intersects with the path of the second robot using the velocity curve obtained from step one.

(3) In Figure 5-10, the rectangular box defined by $[s_i,s'_i]$ and $[t_i,t'_i]$ represents the forbidden region of the velocity curve of the second robot. If more than one robot intersect the path, the forbidden region in the s-t map for each robot crossing the path is computed. By treating the forbidden regions as obstacles, mobility planning becomes a problem of finding a smooth collision-free path in the s-t map. Using the s-t map and a parametric function describing the desired velocity curve, the SDSA algorithm can be used to find a smooth collision-free velocity curve between the starting and destination points.

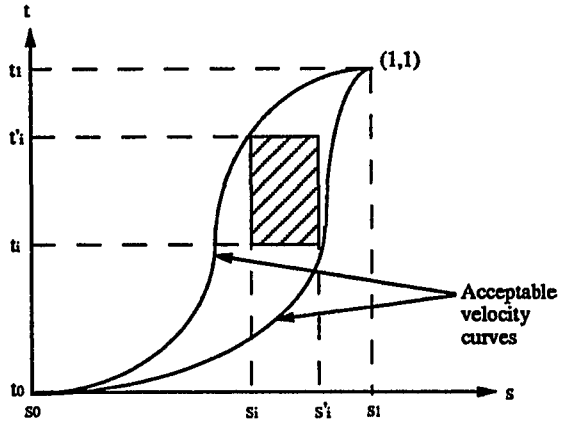


Figure 5-10. Collision-free path planning in the s-t map.

5.8 Conclusions

This chapter presents a Sensor-based Direct Search Algorithm (SDSA) for planning collision-free paths in a natural terrain environment. The significance of this algorithm is threefold. First, it produces paths of any desired degree of smoothness which are readily usable by steering mechanisms of mobile robots. Second, these planned paths minimize the cost of travel. Experiments to find minimum distance collision-free paths have proven the algorithm is useful. Third, this algorithm uses terrain information derived directly from the on-line range data. Therefore, no a priori knowledge about the navigation space is necessary. The SDSA algorithm is also used for planning velocities of the robots moving along the planned paths. Experimental results show that the computation time consumed by the SDSA algorithm is reasonable. It averaged 15 seconds (on 10 trials) to generate a quadratic path (n=m=2) on a terrain map covering a 12 feet by 12 feet square (tested on a Compaq 386/20 personal computer).

Chapter 6

Applications

6.1 Introduction

In the past decade, NASA has sponsored many research projects for building a 3-D computer vision technology base and to apply this technology to space related robotic applications. Robotics researchers at Rice University have developed strong relationships with several groups at the Johnson Space Center in recent years. As a result, the robotics program at Rice has emphasized technologies applicable to space robotics. In support of this research, autonomous robotic systems serving as testbeds for developing algorithms and intelligent control systems for teleoperation space robots are being developed at Rice. These systems consist of an autonomous navigation system, intelligent mobile robot [deFigueiredo 89], and a telepresence control workstation [Cheatham 89b] to provide an interface between the system and human operators. It is hoped that the study of these robotic systems can help us better understand the problems involved in telerobotics.

The ability to navigate autonomously in an unstructured environment is critical to space robots. In exploratory missions to another planet, for example, unmanned vehicles will be used to travel uncharted areas and collect environmental samples. To accomplish this task, space robots must have an on-board, autonomous navigation system. Such an autonomous navigation system, suitable for long range navigation, is proposed in this thesis. The structure of this navigation system is discussed in Section 3 of this chapter. This system is currently being interfaced to the mobile robot (Rice-obot I [deFigueiredo 89]) developed at Rice, and the

experimental results are discussed in Section 4.

In space servicing tasks (such as replacing modular components for satellites), the mobile robots not only must travel in space autonomously, but also are required to interact with the environment in order to accomplish the tasks. An essential, but difficult, subtask is to determine the precise location and orientation of the space objects to be serviced.

At a nominal space station altitude of 270 nautical miles, the sunlight intensity will fluctuate between about sixty minutes of extreme brightness and thirty minutes of near darkness [Bronez 87]. The extreme illumination in space will require special lighting techniques and comprehensive image enhancement algorithms if an optical vision system is used for the positioning task. Using laser scanners to locate satellites also poses problem. Due to the absence of atmosphere light is not diffused or scattered. As a result, viewing object surfaces obliquely will cause large range noise. One solution involves attaching reflective materials to the object. Once the reflective materials have been found, the object can be located. With this information a service robot could accomplish tasks such as replacing the battery pack of a satellite. Section 5 of this chapter describes how a laser range finder (Lasernet by Namco) and an infrared proximity sensors can be used to determine the relative position and orientation of a stationary satellite.

6.2 Rice-obot I

The Rice-obot I is an intelligent mobile robot consisting of a mobile base, two robotic arms, several on-board computers, a stereo vision system, and an ultrasonic system as shown in Figure 6-1 [deFigueiredo 89]. A commercially available mobile base (Labmate by TRC, which contains its own controller and power) is used to provide the mobility of the robot. This base is capable of tracing a smooth trajectory at a user specified speed using the software provided by the company. Two Rhino robot arms, each with five degrees of freedom, are mounted on Z-tables. The Z-tables are vertical platforms which move the arms along the vertical direction, thus adding an additional degree of freedom. The on-board vision system consists of two CCD B/W cameras and several ultrasonic ranging devices. A laser range scanner is being developed to provide additional 3-D imaging capability. A 68030 based computer (made by LYNX Systems) is used as the main on-board processor. This processor runs a real-time version of UNIX especially designed for control of devices such as robotic arms. Two servo controller cards on the VME bus are used to control the motors on the robot, including those on both arms, the Z-tables and the pan/tilt/aim of the cameras. Communication between the mobile robot and the telepresence control workstation (which acts as the user interface) is established via radio frequency RS232 and video transmitters.

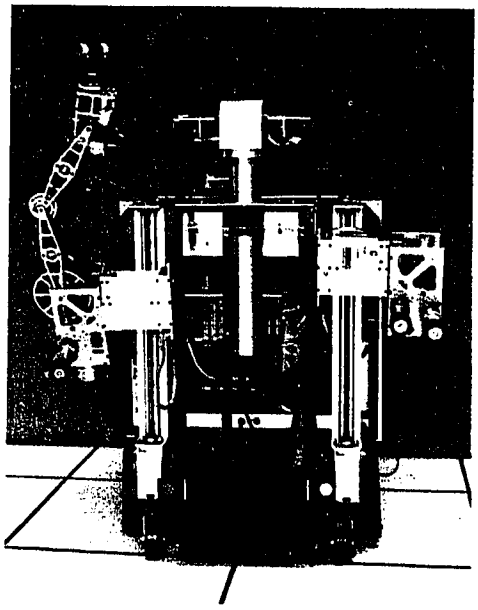


Figure 6-1. Rice-obot I (The left arm was disassembled for maintenance when this photo was taken).

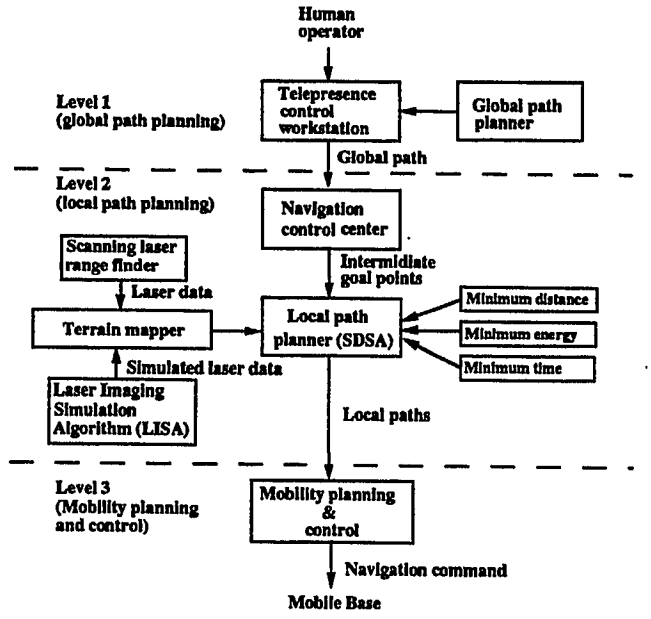


Figure 6-2. Block diagram of the autonomous navigation system.

6.3 The Autonomous Navigation System

As shown in Figure 6-2, the autonomous navigation system consists of three levels of operations: global path planning, local path planning, and mobility planning and control. In the first level of operation, a global path planner (for fully autonomous navigation) or a human operator (for semiautonomous navigation) or both are used to produce a long range path for the mobile robot. In the exploratory missions described in the previous section, for example, low resolution satellite images of the planet's surface may be available to the astronauts. A global path of special interest can be chosen from the satellite images. Due to low resolution, however, each point in the satellite image actually represents a small surface area. Moving from one point to the next on the global path physically means *jumping* from one area to another. Hence a lower level of path planning is needed. This is accomplished by using a local or dynamic navigation planner in the second level of path planning. Given the intermediate goal points along the global path, the local path planner generates safe routes from the current position to the next intermediate goal point using local maps generated from on-line sensory data. The SDSA algorithm and the Hughes path planner discussed in the preceding chapter are used as the local path planners. A second order path (path can be described by a quadratic parametric equation) will be searched using the SDSA algorithm. The simplex searching algorithm is used for finding the optimal path. If the simplex algorithm fails to find a second order path after a number of iterations, higher order paths will then be searched. If a smooth path does not exist (this is concluded from the algorithm if the total number of iterations exceeds a preset value and a

smooth path is still not found), the Hughes path planner will then be used to find a line-segmented path. In the third level of operation, mobility planning and control, motion commands are issued to the mobile robot for executing the navigation plan produced by the first two levels.

The navigation system developed in this research uses an open loop structure. A navigation task is accomplished by executing these three levels of operation sequentially (as discussed in Chapter 2). Smoother operation can be obtained by employing a feedback loop structure. With the feedback structure, the three levels of operation are executed simultaneously; the local path planner computes the next path using the *look ahead* sensory data while the mobile robot executes the current path and the operator (first level planner) monitors the whole operation in the telepresence control workstation while planning the next task for the mobile robot.

A TI Explorer II workstation and an IBM-AT personal computer are used by the navigation system. The Explorer is currently used for planning the global path and is planned to be used for handling most of the high-level artificial intelligence tasks in the future developments. A map-guided path planner developed and implemented on the Explorer by Meng [Meng 87] provides the global path planner. The SDSA algorithm running on the AT computer is used as the local path planner. Communication between the TI explorer and the AT is established using the PC/TCP communication software developed by FTP Software Inc. Planned paths are transmitted to Rice-obot via the radio frequency RS232.

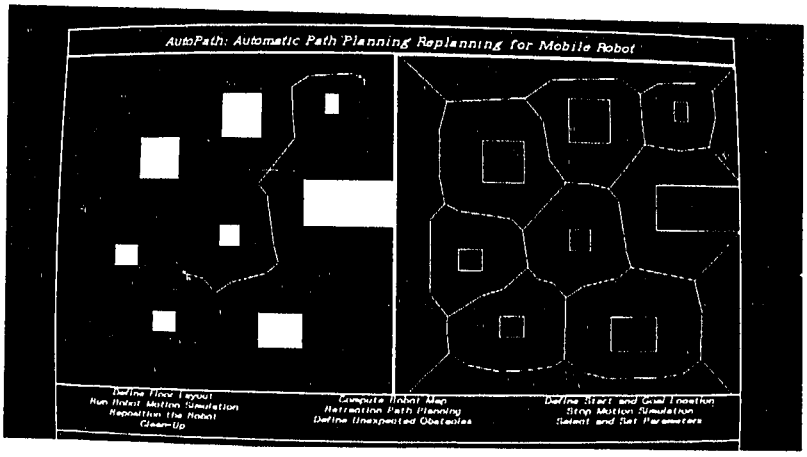
6.4 Application to Autonomous Indoor Navigation

The autonomous navigation system described above is currently being interfaced to the Rice-obot. Since the development of the Rice laser scanner is still in progress, each subsystem described above has been tested successfully in a piecewise manner. Data generated from each subsystem are transferred manually to the next operation.

In the experiment, a 25 feet by 25 feet room filled with randomly placed obstacles (as shown in Figure 6-3a) was used to test the navigation system. The Rice-obot is required to move from the current position to a destination specified by the human operator. The global path planner employed by the navigation system is similar to the graph searching algorithm described in Section 5.1. However, it produces paths along the center lines of the free spaces (as shown in Figure 6-3b) rather than connecting the vertices of polygons that enclose the obstacles. Intermediate goal points, each 10 feet apart, are obtained from the global path. For each pair of intermediate goal points, a local path is generated using laser data generated by computer simulation program described in Chapter 3. The simulation data will be replaced with actual laser data when the on-board scanner is available. Figure 6-4 shows the simulated laser image of the area covering the first 10 feet of the global path (shown in Figure 6-3a) and Figure 6-5 shows the local path generated by the SDSA algorithm.

Once the local path is generated, it is transferred to Rice-obot for execution. To keep the experiment simple, the Rice-obot was commanded to move along the planned path at a constant speed. Continuing efforts will be directed to combine

these subsystems into a whole unit to provide a successive operation. A more complex velocity profile of the mobile robot will also be considered in the future.



(a) (b)

Figure 6-3. (a) Global path generated by the Meng's Algorithm, (b) center lines of the free spaces in the test environment.

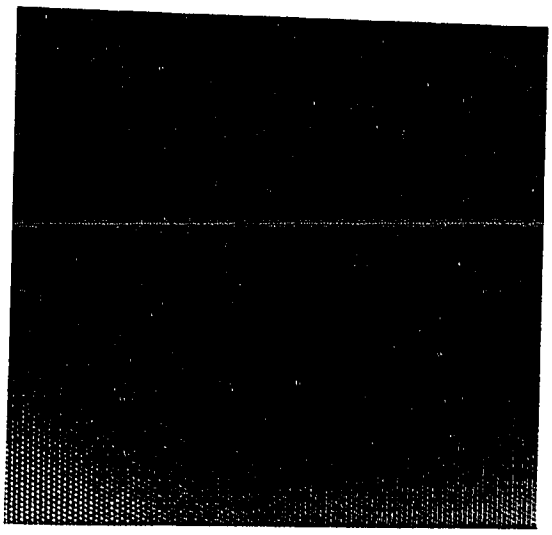


Figure 6-4. Range data generated by LISA.

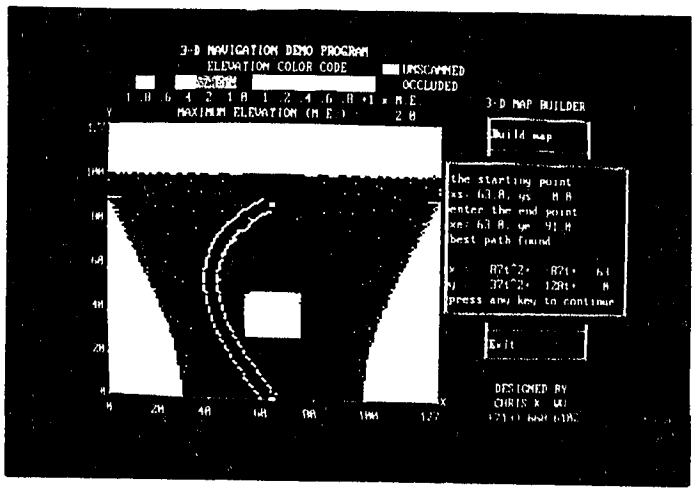


Figure 6-5. Local collision-free path generated by the SDSA algorithm.

6.5 Battery Replacement in a Satellite Model

The objective of the second experiment was to develop a procedure for changing modular components in a stationary satellite. An essential part of this experiment is to locate a satellite model without using a vision system. Consider a simple task where the battery pack of a satellite housed in the space station needs to be replaced: assuming that the satellite is within the robot's work space. However, precise position and orientation of the satellite relative to the robot are unknown. The following paragraphs discuss how a PUMA 560 robot (see Figure 6-6) equipped with a scanning laser range finder (Lasernet by Namco), infrared proximity sensor, and JR3 6-axis force/moment transducer was used to accomplish this task.

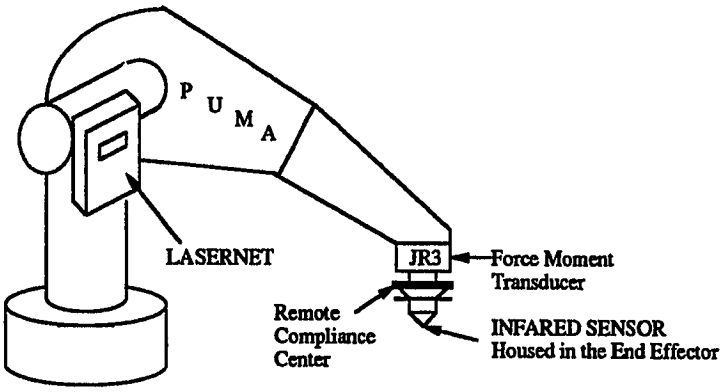


Figure 6-6. PUMA 560 with intelligent sensors.

A Namco Lasernet scanner is attached to the first joint of the PUMA 560 robot. It is employed for both the operations of target detection and range and angle determination. This scanner sends a single narrow beam of light across a

vertical search of 90° out to a distance of 20 feet (see Figure 6-7). Rotating this arc about the PUMA's first joint effectively provides a complete search of the PUMA's workspace. Light that strikes a specially designed reflective tape is returned and sensed by a photo detector in the Lasernet. The return pulse is used to determine both angle and distance measurements and to provide a binary target in view signal. Commercially available reflective tapes have been observed reflecting incident light back to the origin at angles up to 70 degrees. To determine the distance, the width of the target must be known for the Lasernet. Ideally, a spherical target will provide the most uniform projection at any orientation. In this experiment, a spherical reflective target was constructed out of the reflective tape adhered to a golf ball.

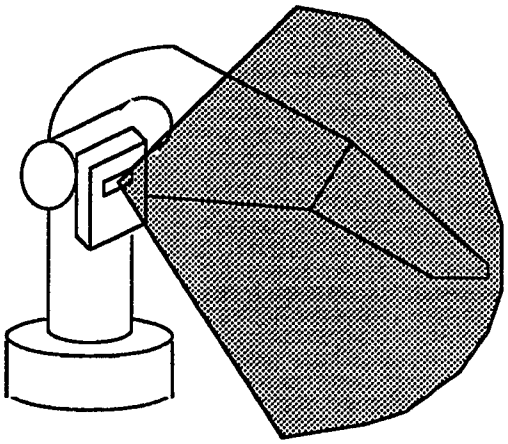


Figure 6-7. Field of view of the Lasernet laser scanner.

The proximity sensor, which is housed in the end effector (see Figure 6-8), consists of an infrared light source and a photodetector. When the infrared light reflected from a surface is detected by the photodetector, a signal is generated to

indicate the presence of the detected surface. The effective distance (to trigger the photodetector) between the sensor and the surface is dependent upon the reflectivity of the subject surface; the highly reflective tape used for the Lasernet target can be sensed at distances of up to 1 foot.

During operation, a model of a satellite with its battery compartment (see Figure 6-9) parallel to the horizontal plane was used. The PUMA rotates the Lasernet about its first joint searching for the spherical target attached to the satellite. When the target is located the Lasernet sends a signal to the PUMA which records its first joint angle. The location of the spherical target is then computed from the range and angle data [Cheatham 88a]. Once the spherical reflector has been located, the proximity sensor will locate another smaller reflective target attached to the object at a known distance. The end effector conducts a circular search of a radius equal to the known distance between the two targets about the first target. By locating the two targets, a line is defined on which the satellite is located.

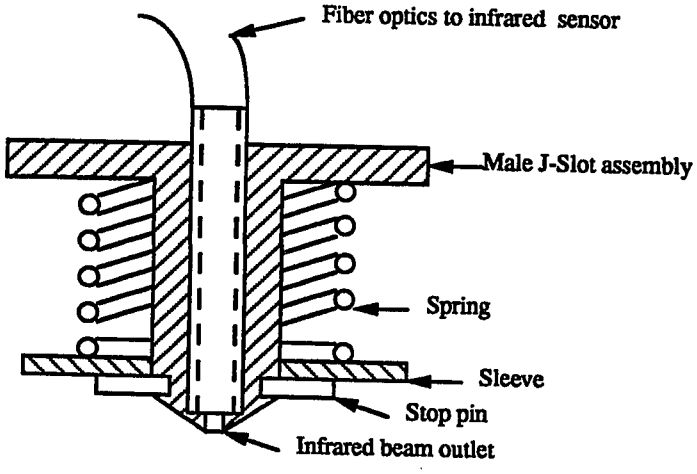


Figure 6-8. Infrared sensor housed in an interchangeable end effector.

The proximity sensor is then lowered to detect the satellite's surface. It must be within 1 inch of the brushed aluminum surface to sense the surface. Once the surface is located, then sensor searches for the edges of the satellite to determine its orientation. The sensor receives a positive impulse while it is over the surface, but will lose the signal once it crosses over the edge. The orientation of the satellite is found by tracing one of the edges. Once an edge is found, the next step is to determine which edge it is (i.e. right or left). This is accomplished by using a false edge (see Figure 6-9). A non-reflective strip is placed along the length of the object. When first seen by the sensor, it will appear as an edge. The sensor will continue onward after finding the edge's other side and regain contact with the surface. This "on-off-on" signal signifies the location of the false edge.

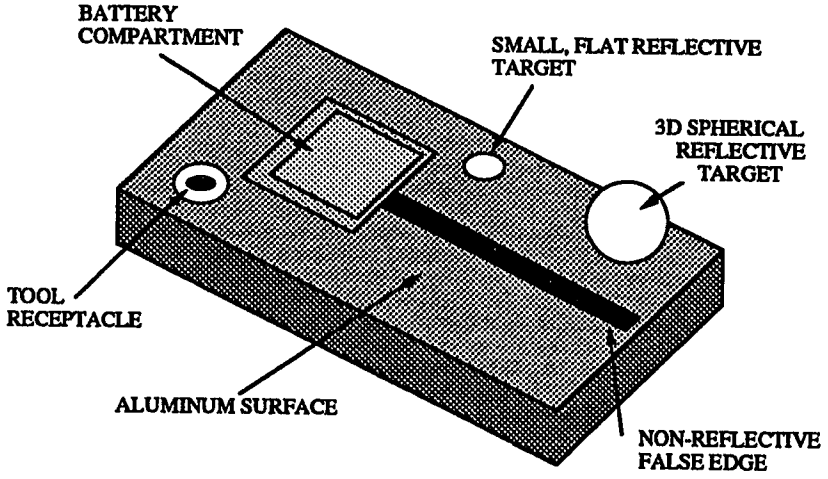


Figure 6-9. Front panel of the battery compartment.

After locating the satellite model, the robot can begin replacing the battery.

The PUMA carries a tray with tools and fresh batteries in a position known to the robot. It engages an electric cover actuator to turn a power screw which opens and closes the door over the battery compartment. A J-Slot type assembly is used to pick up both the battery packs and the electric cover actuator [Armstrong 87]. Figure 6-10 depicts the electric actuator with the J-Slot. The retrieval and insertion of batteries into the battery compartment are performed under position and force control using the JR3 sensor [Cheatham 88b].

The full capabilities of the system were applied to the target object situated at varying heights and arbitrary orientations, primarily normal to the horizontal plane. The described operations were reliable and repeatable for various configurations. However, objects not orthogonal to the horizontal plane required special consideration. The type of application and degree of acuteness to the normal determines which procedure to use [Cheatham 88b].

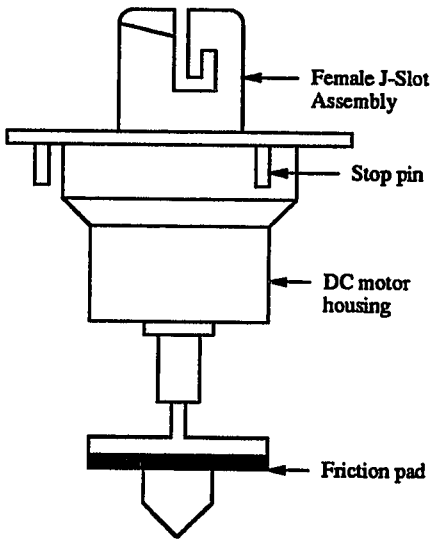


Figure 6-10. Electric actuator with J-slot assembly.

6.6 Conclusion

This chapter describes an autonomous navigation system suitable for long range navigation and a procedure for determining the location and orientation of an object using a Lasernet scanner and an infrared proximity sensor. This navigation system has two advantages over the other existing systems (discussed in Section 5.2):

(a) A global path is derived using reduced knowledge about the environment. Only the significant features of the environment are modeled for planning the global path. This feature significantly reduces the required processing time for generating the global path.

(b) If the global path passes through any obstacle that is not modeled in the global environment, the obstacle will be detected (through traversability test) by the local path planner. Smooth collision-free local paths will be generated using sensory data generated by the on-board sensors. This feature allows a collision-free navigation in an unconstrained (or dynamic) environment.

For short range navigation (in which the navigation space is within the field of view of the on-board sensor), this system may be applied simply by excluding the global path planner from the system.

In many robotic tasks in which the mobile robot is required to move from its current position to a destination and perform object manipulation. Position error of the mobile robot at the destination due to robot movements is inevitable. For manipulation tasks requiring fine motion operation, precise position of the robot

relative to the object is crucial to a successful operation. The methodology of using intelligent sensors for determining the robot's position relative to the object described in this chapter is ideal for this purpose.

Chapter 7

Summary and Conclusions

7.1 Summary

This thesis presents a simulation of a laser imaging system for automated vehicle guidance and space servicing tasks. Data generated by this system are used to command a robot navigating in an unconstrained terrain. The issues addressed in this thesis consist of four parts: the development of a computer graphics program to emulate laser scanners, techniques to transform and interpret range data into meaningful terrain maps, a 3-D navigation planner, and the use of auxiliary sensors such as the infrared proximity sensor for satellite servicing tasks.

A vision system is used primarily to provide meaningful world information of the current state of the robot environment via interpretation of sensory data. Among those vision systems, laser imaging systems have proven to be the most effective (Chapter 2). Unfortunately, a state-of-the-art laser scanner is very costly. Many laser imaging algorithms have thus been developed and tested on computer generated (C-G) laser data. This thesis develops a Laser Imaging Simulation Algorithm (LISA) to emulate laser scanners such as the ERIM and Odetics scanners (Chapter 3). This algorithm generates C-G laser range and reflectance data and displays system shortcomings of real laser images including wrap around interval, missing points and noise. These features are useful in developing algorithms for use with real laser imaging systems.

An objective of laser image processing is to transform and interpret laser data into meaningful environment information to support the robot's applications.

Procedures for developing topographical terrain maps from range data are described in this thesis (Chapter 4). A noise insensitive edge detector was developed for extracting wireframes of objects from range data. By combining this edge detector with the Locus algorithm developed at Carnegie Mellon University, topographical terrain maps were derived from the sensory data. The terrain maps generated in this research consist of a gridwork of 128 by 128 points. Each grid cell represented a 1.125"x1.125" square area. Grid points in the terrain map are classified into four types: unscanned, scanned, shadow, and invisible. Unscanned points represent the area outside the field of view of the laser scanner, and the scanned points contain elevation values of the terrain surface. Occluded areas are represented by shadow points, and the area occupied by *invisible* objects will be identified by invisible points.

Autonomous robot navigation is becoming an important topic in factory automation and space robotics. In exploratory missions on another planet, for example, unmanned vehicles will be used to travel uncharted areas and collect environmental samples. Navigation involves planning an acceptable route from a starting point to a destination (Chapter 5). Autonomous navigation means to use an intelligent machine without human interference to accomplish this task. There are many navigation algorithms that work fairly well for 2-D navigation. But the real problem is devising algorithms for natural terrain (3-D) environments. This thesis presents a Sensor-based Direct Search Algorithm (SDSA) for planning collision-free paths in a natural terrain environment. The significance of this algorithm is threefold. First, it produces paths of any desired degree of smoothness which are

readily usable by steering mechanisms of mobile robots. Second, these planned paths minimize the cost of travel. Experiments to find minimum distance collision-free paths have proven the algorithm is useful. Third, this algorithm uses terrain information derived directly from the on-line range data; this feature allows the algorithm to be applied to navigation tasks in an unconstrained nature terrain. The SDSA algorithm is also used for planning velocities of robots moving along the planned paths.

Chapter 6 describes an autonomous navigation system suitable for long range navigation and a procedure for determining the location and orientation of an object using a Lasernet scanner and an infrared proximity sensor. The autonomous navigation system decomposes the navigation tasks into three levels of operations: global path planning, local path planning, and mobility planning and control. This navigation system is currently being interfaced to the mobile robot (Rice-obot I) developed at Rice; experiment results have shown the usefulness of this system. The navigation system developed in this research has two advantages over the other existing systems:

(a) A global path is derived using the reduced knowledge about the environment. Only the significant features of the environment are modeled for planning the global path. This feature significantly reduces the required processing time for generating the global path.

(b) If the global path passes through any obstacle that is not modeled in the global environment, the obstacle will be detected (through traversability test) by the local path planner. Smooth collision-free

local paths will be generated using sensory data generated by the on-board sensors. This feature allows a collision-free navigation in an unconstrained (or dynamic) environment.

Due to the extreme illumination and the lack of atmosphere in space, the use of vision systems for space servicing tasks (such as replacing modular components for satellites) are not desirable (Chapter 6). An essential but difficult subtask in space robotics is thus to determine the location and orientation of the space objects without using a vision system. The thesis describes a methodology of using a Lasernet scanner and an infrared proximity sensor to determine the robot's position relative to the object. A simple task of replacing the battery pack of a satellite model has been successfully executed in the experiment. This system provides a fairly inexpensive augmentation to a robotic system to improve the robot's versatility.

7.2 Conclusions and outlook for the future research

The developments of laser imaging algorithms can be achieved using real or computer generated laser data. Computer generated data are easier, less expensive, and provides precise control of the positions of the laser scanner and simulated objects. Laser imaging algorithms developed using the simulated laser data demonstrate the usefulness of the simulated data. This thesis argues that the laser imaging system is an excellent system for automated vehicle guidance and space servicing tasks. The navigation system developed here suggests that a fully autonomous navigation system will be feasible in the near future.

Some fundamental issues in space robotics were not addressed in this thesis, but could be investigated as extension of this thesis.

(a) *System Integration.* System integration is an essential (perhaps the most important) problem in developing a large scale robotic system. It involves developments of the system architecture (capable of combining all of the control, sensory, and knowledge subsystems of the intelligent robotic systems), communication between subsystems, and data base management within subsystems. A well planned integration scheme will not only minimize the design efforts required to accomplish the task (and therefore minimize the cost) but also increase the possibility of a successful task.

(b) *Mobility Planning and Control.* In this thesis, mobility planning and mobile base control were solved as two independent problems. This approach will not pose any problem as long as the velocity of mobile robot is maintained within a relatively slow speed. Nevertheless, if maximum speed along the planned path is desired, mobility planning and control must be considered as a coupled problem. Extended research should be continued on developing a mobility planning algorithm which incorporates the physical constrains of the control hardware.

(c) *Path Planning for Free-Flying Robots.* In the final phase of the development of space robots, robots are made fully autonomous and capable of free-flying for orbiter/satellite inspection, retrieval, and

repair tasks. To accomplish this task, world models of the deep space environment and algorithms for collision-free navigation in the deep space need to be researched. Another essential component to accomplish this task is the need of a very fast computing machine since space objects to be serviced in this case are also in motion.

References

Acharya, P. K. and Henderson, T. C., "Parameter Estimation and Error Analysis of Range Data", *International Conference on Pattern Recognition*, 1988, pp. 1709-1714.

Aggarwal, J. K., et al., "Survey: Representation Methods for 3-D Objects," *Progress in Pattern Recognition*, North-Holland Publishing Company, 1981, pp. 377-391.

Aggarwal, J. K. and Magee, M. J., "Determining Motion Parameters Using Intensity Guided Range Sensing". *Pattern Recognition 19*, Vol. 2 (1986). pp. 169-180.

Aleksander, I., Stonham, T. J. and Wilkie, B. A., "Computer Vision System for Industry: Comparisons", *Artificial Vision for Robots* (edited by Aleksander, I.), Chapman and Hall Publishing Company, New York, 1984, pp. 179-196.

Allen, G., "An Introduction to Solid Modeling", *Computer Graphics World*, v. 11, 1982, pp. 32-38.

Allen, P. K., *Robotic Object Recognition Using Vision and Touch*, Kluwer Academic Publishers, Boston, 1987.

Araki, K., Sato, Y., Tanaka, N., and Fujino, T., "A Method for High Speed 3-D Range Measurement and Its Trial Instrumentation", *Proc. 9th International Conf. on Pattern Recognition*, Nov. 1988, Rome, Italy, pp. 755-757.

Arkin, R. C., "Navigational Path Planning For a Vision-based Mobile Robot", *Robotica International Journal of Information, Education and Research in Robotics and Artificial Intelligence*, 1988, pp. 49-63.

Arkin, R. C., "Motor Schema-Based Mobile Robot Navigation", *The International Journal of Robotics Research*, 1989, pp. 92-112.

Armstrong, A., Dunn, A., Hekma-Wierda, D., Hodge, A., and Howard, D., "Design, Construction and Implementation of a Robotic System for Replacing Batteries", *Senior Design Project Report*, MEMS Dept., Rice University, May, 1987.

Asada, M., "Building a 3-D World Model for a Mobile Robot from Sensory Data", *IEEE Proc. International Conference on Robotics and Automation*, 1988, pp. 918-923.

Asada, M. and Tsuji, S., "Shape From Projecting a Stripe Pattern", *IEEE Proc. of International Conference on Robotics and Automation*, 1987, pp. 787-792.

Ballard, D. H. and Brown, C. M., *Computer Vision*, Prentice-Hall, Englewood Cliffs, N.J., 1982.

Besl, P. J. and Jain, R. C., "Three-Dimensional Object Recognition", *ACM Computing Surveys*, Vol. 17 (1), March, 1985, pp. 75-145.

Besl, P. J. and Jain, R. C., "Segmentation through Symbolic Surface Descriptions", *Proc. Computer Vision and Pattern Recognition*, Miami, 1986, pp. 77-85.

Besl, P. and Jain, R., "Range Image Understanding," *IEEE Proc. Computer Vision and Pattern Recognition*, San Francisco, June 1985, pp. 430-449.

Beyer, J., Jacobus, C., and Mitchell, B., "Orbital Navigation, Docking, and Obstacle Avoidance As a Form of Three Dimensional Model-Based Image Understanding", pp. 37-46.

Bhanu, B., Lee, S., Ho., C., and Henderson, T., "Range Data Processing: Representation of Surfaces by Edges", *Proceedings of Eighth International Conference on Pattern Recognition*, Paris, France, Oct. 1986, pp.236-238.

Brady, J. P., Nandhakumar, N., and Aggarwal, J. K., "Recent Progress in the Recognition of Objects from Range Data", *IEEE 9th International Conference on Pattern Recognition*, Rome, Italy, Vol. 1, Nov. 1988, pp. 85-92.

Bronez, Mark A., Vision Concepts for Space-Based, *Tech Report SSS 86-0165, Rockwell International*, Space Station Systems Division, Downey, California, January, 1987.

Cahn, D. F. and Phillips, S. R., "ROBNAV: A Range Based Robot Navigation and Obstacle Avoidance Algorithm", *IEEE Trans. on Systems, Man and Cybernetics*, Vol. SMC-5 (5), Sep., 1975, pp. 544-551.

Chattergy, R., "Some Heuristic for the Navigation of a Robot", *The International Journal of Robotics Research*, Vol. 4, No. 1, Spring 1985, pp. 59-65.

a. Cheatham, J. B., Weiland, P. L., and Wu, C. K., "A Sensor System for Determining Position and Orientation of Robot Targets", *3rd International Conference on CAD/CAM Robotics and Factories of the Future*, Southfield, MICH, 1988.

b. Cheatham, J. B., Wu, C. K., Weiland, P. L., and Cleghorn, T. F., "A Multi-Sensor System for Robotics Proximity Operations", *Space Operation Automation and Robotics (SOAR 88), Second Annual Workshop on Automation and Robotics*, Dayton, Ohio, July, 1988.

c. Cheatham, J. B., Wu, C. K., Chen, Y. C., and Cleghorn, T. F., "Interpretation of Robot Vision System Images via An Expert System", *Proc. ASME international Computers in Engineering Conference*, San Francisco, CA, July, 1988.

a. Cheatham, J. B, Wu, C. K., Weiland, P. L., and Norwood, J. D., "Framework for A Laser Navigation System", *Research Institute for Computing and Information Systems Symposium 1989*, Univ. of Houston-Clear Lake, 1989.

b. Cheatham, J. B., "A Telepresence Space Robot Prototype", *Proposal to the Advanced Research Program/Advanced Technology Program*, #003604003, 1989.

Daily, M., Harris, J., Keirsey, D., Olin, K., Payton, D., Reiser, K., Rosenblatt J., Tseng, D., and Wong, V., "Autonomous Cross-Country Navigation With the ALV", *IEEE Proc. International Conference on Robotics and Automation*, 1988, pp. 718-726.

Daily, M. J., Harris J. G. and Reiser, K., "Detecting Obstacles in Range Imagery", *Proc. Image Understanding Workshop*, Feb., 1987, pp. 87-97.

de Figueiredo, R. J. P., "A Framework for Automation of 3D Machine Vision," *Texas Instruments Technical Journal, Special issue on Applications of Artificial to Industrial Automation*, Winter 1987, pp. 62-72.

de Figueiredo, R., Ciscon, L., and Berberian, D., "RICE-OBOT I: An Intelligent Autonomous Mobile Robot", *NASA Conference on Space Telerobotics*, Pasadena, CA, Jan 29, 1989.

Drumheller, M., "Mobile Robot Localization Using Sonar", *MIT AI Lab Memo AIM-824*, 1985.

Duda, R. O., Nitzan, D., and Barrett, P., "Use of Range Reflectance Data to Find Planar Surface Regions", *Trans. Pattern Analysis and Machine Intelligence*, Vol. 1, No. 3, July 1979, pp. 259-271.

Duda, R. O. and Hart, P. E., *Pattern Classification and Scene Analysis*, John Wiley and sons, New York, 1973.

Dunlay, R. T.,"Obstacle Avoidance on Roadways Using Range Data", *SPIE-Mobile Robots*, 1986, pp. 110-116.

Dunlay, R. T., "Obstacle Avoidance Perception Processing for The Autonomous Land Vehicle", *IEEE Proc. International Conference on Robotics and Automation*, New York, 1988, pp. 912-917.

Elfes, A., "A Sonar-Based Mapping and Navigation System", *IEEE Proc. of International Conference on Robotics and Automation*, 1986, pp. 1151-1156.

Flynn A. M. and Brooks, R. A., "MIT Mobile Robots - What's Next", *IEEE Proc. International Conference on Robotics and Automation*, 1988, pp. 611-617.

Foley, J. D. and Van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing company, Massachusetts, 1982.

Gaw, D. and Meystel, A., "Minimum-time Navigation of an Unmanned Mobile Robot in a 2-1/2D World with Obstacles", *IEEE International Conference on Robotics and Automation*, 1986, pp. 1670-77.

Gilmore, J. F., Semeco, A. C., and Eamsherangkoon, P., "A Heuristic Route Planner for Autonomous Robots", *SPIE Intelligent Robots and Computer Vision*, Vol. 521, 1984, pp. 160-68.

Gonzalez, R. C. and Wintz, P., *Digital Image Processing*, Addison-Wesley Publishing Company, Massachusetts, 1987.

Gordon, S. J. and Seering W. P., "Locating Polyhedral Features From Sparse Light-Stripe Data", *IEEE Proc. of International Conference on Robotics and Automation*, 1987, pp. 801-806.

Goto, Y. and Stentz, A., "The CMU System for Mobile Robot Navigation", *IEEE Proc. International Conference on Robotics and Automation*, 1987, pp. 99-105.

Harrington, S., *Computer Graphics: A Programming Approach*, McGraw-Hill Book Company, New York, U.S.A., 1983.

Hearn, D. and Baker, M. P., *Computer Graphics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

Hebert, M., Caillas, C., Krotkov, E., Kweon, I. S., and Kanade, T., "Terrain Mapping for a Roving Planetary Explorer", *IEEE International Conference on Robotics and Automation*, Scottsdale, Arizona, May, 1989, pp. 997-1002.

Hebert, M. and Ponce, J., "A New Method For Segmenting 3-D Scenes Into Primitives", IEEE, 1982, pp. 836-38.

Hebert, M., Kanade, T., and Kweon, I., "3-D Vision Techniques for Autonomous Vehicles", *Technical Report CMU-RI-TR-88-12*, Carnegie Mellon University, The Robotics Institute, 1988.

Hooke, R. and Jeeves, T. A., " 'Direct Search' Solution of Numerical and Statistical Problems", pp. 212-29. ~1960

Horn, B. K. P., *Robot Vision*, McGraw-Hill, New York, 1986.

Hwang, Y. K. and Ahuja, N., "Path Planning Using a Potential Field Representation", *IEEE International Conference on Pattern Recognition*, 1988, pp. 648-49.

Ikeuchi, K. and Horn B. K. P., "Numerical Shape from Shading and occluding boundaries", *Artificial Intelligence 17*, Aug., 1981, pp. 141-184.

Inokuchi, S., Sato, K., and Matsuda, F., "Range-Imaging System for 3D Object Recognition", IEEE, 1984, pp. 806-8.

Inokuchi, S., Nita, T., Matsuda, F. and Sakurai, Y., "A Three Dimensional Edge-region Operator for Range Picture", *Proc. Int. Conf. on Pattern Recognition*, 1982, pp. 918-920.

Iyengar, S. S., Jorfensen, C. C., Rao, S. V. N. and Weisbin, C. R., "Robot Navigation Algorithms Using Learned Spatial Graphs", *Robotica*, Vol. 4, 1986, pp. 93-100.

Jarvis, R. A., "A Perspective on Range Finding Techniques for Computer Vision", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-5, No. 2, March 1983, pp. 122-138.

Jarvis, R. A. and Byrne, J. C., "An Automated Guided Vehicle with Map Building and Path Finding Capabilities", *Robotics Research*, MIT Press, 1987, pp. 497-504.

Kanade, T. (editor), *3-D Machine Vision*, Kluwer Academic Publishers, 1987.

Kant, K. and Zucker, S. W., "Planning Smooth Collision-Free Trajectories: Path, Velocity and Splines in Free-Space", *International Journal of Robotics and Automation*, Vol. 2, No. 3, 1987, pp. 117-124.

Khatib, O., "Real-time Obstacle Avoidance for Manipulators and Mobile Robots", *The International Journal of Robotics Research*, Vol. 5, No. 1, Spring 1986, pp. 90-98.

Krishen, K. and Defigueiredo, R. J. P., "Vision Technology/ Algorithms for Space Robotics Applications", *SOAR First Annual Workshop on Space Operations Automation and Robotics*, Houston, TX, Aug. 1987.

Krotkov, E. and Martin, J. P., "Range From Focus", CH2282-2/86/0000/1093, 1986 IEEE, pp. 1093-1098.

Laurendeau, D., Poussart, D., "A segmentation algorithm for extracting 3D edges from range data", *Proceedings of COMPINT 85: Computer Aided Technologies*, pp 765-7, Montreal, Que., Canada, Sept. 1985.

Liu, Y. H., Kuroda. S., Naniwa, T., Noborio, H., and Arimoto, S., "A Practical Algorithm for Planning Collision-Free Coordinated Motion of Multiple Mobile Robots", *IEEE Proc. of International Conference on Robotics and Automation*, Scottsdale, Arizona, May 1989, pp. 1427-1432.

Lozano-Perez, T., "Spatial Planning: A Configuration Space Approach", *IEEE Transactions on Computers*, Vol. C-32, No. 2, Feb 1983, pp. 108-20.

Luenberger, D. G., *Linear and Nonlinear Programming*, 2nd edition, Addison-Wesley Publishing Co., Massachusetts, 1984.

Matthies L. and Elfes, A., "Integration of Sonar and Stereo Range Data Using a Grid-Based Representation", *IEEE Proc. International Conference on Robotics and Automation*, 1988, pp. 727-733.

Meng, A., "A Methodology of Map-Guided Autonomous Navigation with Range Sensor in Dynamic Environment", *Proc. of the SPIE Conference on Mobile Robotics III*, 1987.

Mitiche, A. and Aggarwal, J. K., "Detection of Edges Using Range Information", *Trans. on Pattern Analysis and Machine Intelligence*, Vol. 5, No. 2, Mar., 1983, pp. 174-178.

Monaghan, G. E., " World Modeling and Path Planning for Autonomous Mobile Robots", *SPIE Intelligent Robots and Computer Vision*, Vol. 579, 1985, pp. 390-97.

Moon, p., *The Scientific Basis of Illuminating Engineering*, Dover Publications, New York, 1961.

Moravec, H. P., "The Stanford Cart and the CMU Rover", *Proceedings of the IEEE*, 71, July, 1983. pp. 872-884.

Nelder, J. A. and Mead, R., "A Simplex Method for Function Minimization", *The Computer Journal*, Vol. 8, No. 1, April 1985, pp. 308-313.

Nelson, W., "Continuous-Curvature Paths for Autonomous Vehicles", *IEEE International Conference on Robotics and Automation*, Scottsdale, Arizona, May, 1989, pp. 1260-64.

Newman, W. M. and Sproull, R. F., *Principles of Interactive Computer Graphics*, McGraw-Hill Book Company, New York, U.S.A., 1979.

Newman, W. S., "High Speed Robot Control and Obstacle Avoidance Using Dynamic Potential Functions", IEEE, 1987, pp. 14-24.

Nitzan, D., Brain, A. E., and Duda, R. O., "The Measurement and Use of Registered Reflectance and Range Data in Scene Analysis", *Proc. of the IEEE*, Vol. 65, No. 2, Feb., 1977, pp. 206-220.

Nitzan, D., "Scene Analysis Using Range Data", *Artificial Intelligence Center, Stanford Research Institute*, Menlo Park, CA, Tech. Note 69, Aug. 1972.

Nitzan, D., "Three-Dimensional Vision Structure for Robot Applications", *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 10, No. 3, May 1988, pp. 291-309.

Odetics Inc., *Laser Scanner, Miro-Vax DMA Interface Specification*, Anaheim, CA, Feb. 9, 1987.

Olsson, D. M., "A sequential Simplex Program for Solving Minimization Problems", *Journal of Quality Technology*, 6, 1, 1974, pp. 51-57.

Oommen, B. J., Sitharama Iyengar, S., Rao, N. S. V., and Kashyap, R. L., "Robot Navigation in Unknown Terrain Using Learned Visibility Graphs - I; The Disjoint Convex Obstacle Case", *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 6, Dec. 1987, pp. 672-680.

Parnis, P. J. and Drazan, P. J., "Recognition of Unreliable Ultrasonic Range Data in a Robotic Environment", *Robotica*, Vol. 7, 1989, pp. 183-189.

Pratt, W. K., *Laser communication systems*, John Wiley and Sons, New York, 1969.

Quek, F. K. H., Franklin, R. F., and Pont, F., "A Decision System for Autonomous Robot Navigation Over Rough Terrain", *SPIE Intelligent Robots and Computer Vision*, Vol. 579, 1985, pp. 377-89.

Requicha, A. A. G. and Voelcker, H. B., "Solid Modeling: A Historical Summary and Contemporary Assessment", *Computer Graphics and Applications*, 2, 3, Mar. 1982, pp. 9-26.

Rosenfeld, A. and Kak, A. C., *Digital Picture Processing*, 2nd Edi., Academic Press, Inc., New York, 1982.

Salman, R. and Slater, M., *Computer Graphics: Systems and Concepts*, McGraw-Hill Book Company, New York, U.S.A., 1987.

Shafer, S. A., Stentz, A., and Thorpe, C. E., "An Architecture for Sensor Fusion in Mobile Robot", *IEEE Proc. of International Conference on Robotics and Automation*, 1986, pp. 2002-2011.

Shirai, Y. and Suwa, M., "Recognition of polyhedrons with a range finder", *Proc. 2nd Int. Joint Conf. Artificial Intelligence*, London, England, Sep., 1971, pp. 80-87.

Sorensen, B., Donath, M., Yang, G. B., and Starr, R., "Performance of the Minnesota Scanner: A Prototype for 3D Tracking of Moving Body Segments", IEEE, 1987, pp. 793-800.

Sorensen, B., Donath, M., Yang, G. B., and Starr, R., "The Minnesota Scanner: A Prototype Sensor for Three-Dimensional Tracking of Moving Body Segments", IEEE, 1989, pp. 499-509.

Stenstrom J. R. and Connolly, C. I., "Building Wireframes from Multiple Range Views", *IEEE Proc. of International Conference on Robotics and Automation*, 1986, pp. 615-620.

Sutherland, I. E. and Hodgman, G. W., "Reentrant Polygon Clipping", *Communications of the ACM*, 17(1), Jan. 1974, pp32-42.

Svetkoff, D. J., Leonard, P. F., Sampson, R. E., and Jain, R., "Techniques for Real-Time, 3D, Feature Extraction Using Range Information", *SPIE Intelligent Robots and Computer Vision*, Vol. 521, 1984, pp. 302-9.

Thorpe, C., Hebert, M. H. and Kanade, T., "Vision and Navigation for the Carnegie-Mellon Navlab", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. 10 (3), May, 1988, pp. 362-373.

Weiland, P. L., *The Use of Scanning Laser Range Finders for Autonomous Robotic Navigation*, Master Thesis, Mechanical Engineering Department, Rice University, Houston, Texas, 1990.

Wendlandt, W. W. and Hecht, H. G., *Reflectance Spectroscope*, Interscience Publishers, New York, 1966.

Winston, P. H., *Artificial Intelligence*, 2nd edition, Addison-Wesley Publishing Co., Massachusetts, 1984.

Wu, C. K., Weiland, P. L., and Cheatham, J. B., "A Computer Graphics Testbed for Developing and Testing Laser Imaging Algorithm", *SPIE/SPSE Symposium on Electronic Imaging: Science and Technology*, 1990.

Zaharakis, S. C. and Guez, A., "Time Optimal Navigation Via Slack Time Sets", IEEE, 1988, pp. 650-51.

Zuk, D., Pont, F., Franklin, R., and Larrowe, V., "A System for Autonomous Land Navigation", *Technical Report IR-85-540*, Environmental Research Institute of Michigan, Ann Arbor, MI, 1985.

Appendix A

3-D Perspective Projection

A.1 Introduction

3-D perspective projection is a well developed procedure in computer graphics. Many computer graphics books offer excellent discussions on this topic [Newman 79, Foley 82, Harrington 83, Hearn 86, Salman 87]. This appendix emphasizes only the specific techniques that are used by the laser simulation program discussed in Chapter 3.

As shown in Figure A-1, 3-D perspective projection is accomplished in four steps: normalization, clipping, scaling, and perspective projection. Given the specification of a view volume, perspective projection is done by clipping the objects against the view volume and then projecting the clipped objects onto the view plane. However, clipping against an arbitrary view volume is a complex and computationally expensive process. To alleviate this, normalization makes the clipping and perspective projection become straightforward operations in the normalized coordinate system. The following sections discuss each step in greater detail.

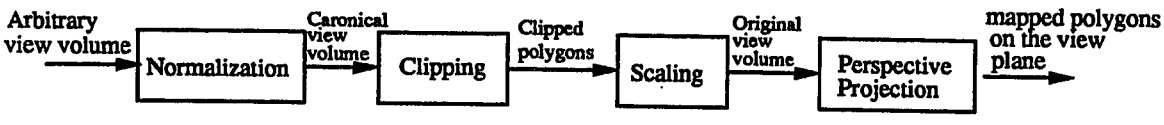


Figure A-1. Block diagram of perspective projection.

A.2 Normalization

The normalization process transforms world coordinate positions into left-handed Cartesian coordinates (local coordinates) attached to the laser scanner and scales the arbitrary pyramid view volume into a canonical one. As shown in Figure A-2, the local coordinates are defined by the position vector **c** of its origin (the view point) and three angles defining its orientation. The unit vector of its z-axis (the optical axis of the scanner) is defined by the pitch (p) and yaw (y) angles as shown in Figure A-3. Pitch p is the angle between the unit vector and the horizontal plane (the x-y plane); yaw y is the angle between the positive x-axis and the projection of the unit vector on the horizontal plane. Positive values are assigned to the angles above the horizontal plane and y is measured using the counter-clockwise as positive convention. Before considering the third orientation angle, one should assume that the positive y-axis of the local coordinates is pointing upward. This is often true unless the scanner is operated upside down. The local coordinates are given by

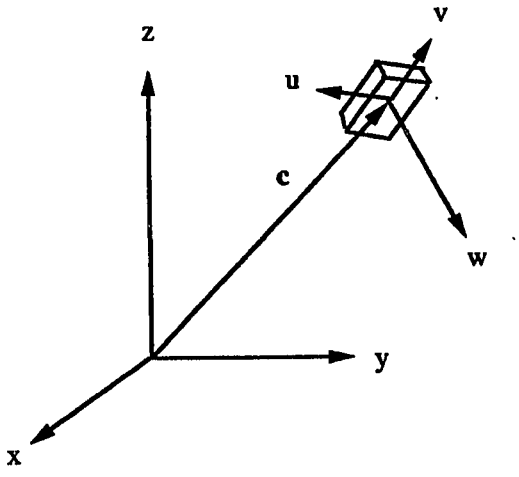


Figure A-2. Coordinate systems.

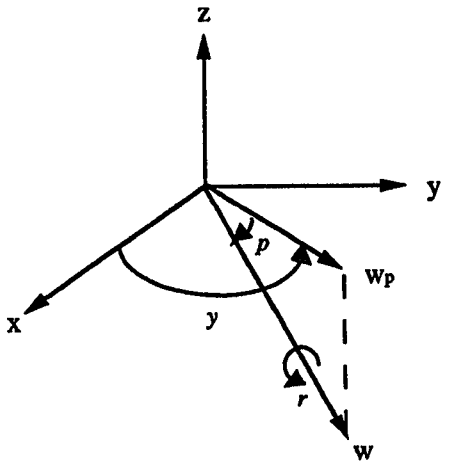


Figure A-3. Pitch, yaw, and roll.

$$\mathbf{w'} = \begin{bmatrix} \cos(p)\cos(y) \\ \cos(p)\sin(y) \\ \sin(p) \end{bmatrix}$$

$$\mathbf{u'} = \mathbf{w'} \times \mathbf{v}_{up}$$

$$\mathbf{v'} = \mathbf{u'} \times \mathbf{w'} \qquad \text{(A.1)}$$

where

$\mathbf{v}^t_{up} = [0\ 1\ 0]$ (if $\mathbf{w'}^t = [0\ 1\ 0]$, then $\mathbf{v}^t_{up} = [1\ 0\ 0]$) and

$\mathbf{u'}$,$\mathbf{v'}$, and $\mathbf{w'}$ are the unit vectors along the x, y, and z axes of the local coordinates before considering the third angle.

The third angle, roll r, represents the scanner's rotation about its optical axis. Given the roll angle, the final position of the local coordinates can be determined by rotating $\mathbf{u'}$ and $\mathbf{v'}$ r degrees about $\mathbf{w'}$. That is

$$\mathbf{w} = \mathbf{w'}$$
$$\mathbf{u} = \mathbf{R}_w \mathbf{u'}$$
$$\mathbf{v} = \mathbf{R}_w \mathbf{v'} \qquad \text{(A.2)}$$

where

$$\mathbf{R}_w = \begin{bmatrix} K_1^2\cos(r)+w_x^2 & K_2w_xw_y+w_z\sin(r) & K_2w_xw_z-w_y\sin(r) \\ K_2w_xw_y-w_z\sin(r) & K_3\cos(r)+w_y^2 & K_4\cos(y)+w_x\sin(r)+w_yw_z \\ K_2w_xw_z\cos(y)-w_z\sin(r) & K_4\cos(y)-w_x\sin(r)+w_yw_z & K_5\cos(r)+w_z^2 \end{bmatrix}$$

and

$$K_1 = w_y^2 + w_z^2$$
$$K_2 = 1 - \cos(r)$$
$$K_3 = \frac{w_x^2w_y^2 + w_z^2}{K_1}$$
$$K_4 = \frac{w_yw_z(w_x^2 - 1)}{K_1}$$

$$K_5 = \frac{w_x^2 w_z^2 + w_y^2}{K_1}$$

if $K_1 = 0$, then $K_3 = K_5 = 1$ and $K_4 = 0$;

r is also measured using the counter-clockwise convention. Given the horizontal (Θ) and vertical (Ψ) scanning angles, a pyramid view volume centered at the optical axis is determined as depicted in Figure A-4. Let **p** be the world position needed to be normalized. If

$$\mathbf{p'} = \mathbf{N}\ \mathbf{p}, \qquad \text{(A.3)}$$

where **p'**is the normalized vector of **p**, how does one find the normalizing transformation **N**? It is achieved by performing the following steps:

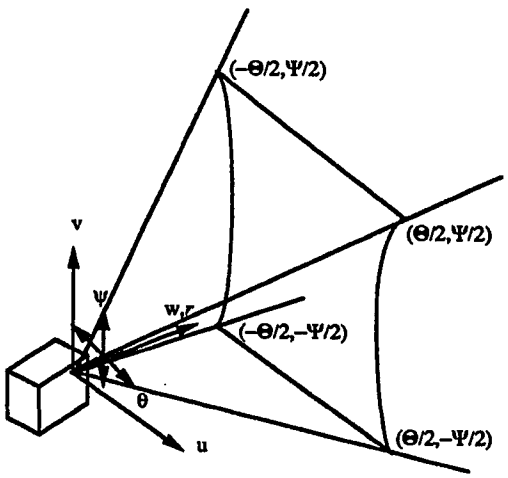


Figure A-4. Pyramid view volume defined by the scanning angles.

(1) Translate the view point to the origin,

(2) Rotate to align the local coordinates to the world coordinates. Since the local coordinate system is a left-handed coordinate system, the z-axis will align with the negative z-axis of the world coordinates,

(3) Change from right-handed to left-handed coordinates,

(4) Scale the view volume into a canonical one.

The first step aligns the origins of both the local coordinates and world coordinates. With **c** as the position vector of the view point, the translation matrix will be

$$\mathbf{P}[-\mathbf{c}] = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(A.4)}$$

where c_x, c_y and c_z denote the x, y, z components of **c**, respectively. Notice that since the translation matrix is defined in a homogeneous coordinate system, the position vectors must be first converted to the homogeneous coordinates.

After calculating unit vectors for the principal axes of the local coordinates (Equations A.1 and A.2), step 2 aligns the coordinate systems by rotation, given by

$$\mathbf{T} = \begin{bmatrix} u_x & v_x & -w_x & 0 \\ u_y & v_y & -w_y & 0 \\ u_z & v_z & -w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(A.5)}$$

The third step converts the world coordinates to a left-handed coordinates by reversing the sign of its z component. That is

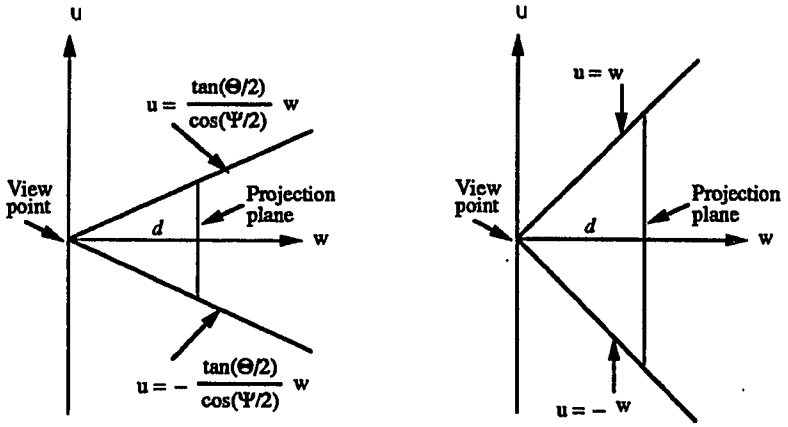
$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(A.6)}$$

The final step scales the x and y axes so that the sloped planes of the view volume will have unity slope. As shown in Figure A-5, the scaling matrix is given by

$$\mathbf{S} = \begin{bmatrix} \dfrac{\cos(\Psi/2)}{\tan(\Theta/2)} & 0 & 0 & 0 \\ 0 & \dfrac{1}{\tan(\Psi/2)} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(A.7)}$$

Combining these transformation matrices, the normalizing transformation is given by

$$\mathbf{N} = \mathbf{S}\ \mathbf{L}\ \mathbf{T}\ \mathbf{P} \qquad \text{(A.8)}$$



(a) Before Scaling (b) After scaling

Figure A-5. Cross section of view volume before and after scaling.

A.3 3-D Clipping Against A Canonical View Volume

Clipping is a procedure of eliminating all parts of the polygon outside the specified view volume. Using the surface boundary representation, a polygon is defined by the position vectors of its vertices. Once its vertices have been normalized, the polygon is ready to be clipped against the canonical view volume. The canonical view volume is defined by five planes: four sloped plane and a front clipping plane. Equations of these planes are given by

Front clipping plane:	$z = 0$
Top clipping plane:	$y = z$
bottom clipping plane:	$y = -z$
Right clipping plane:	$x = z$
Left clipping plane:	$x = -z$

The relations of a point P with the canonical view volume is given by

(1) point is behind view volume if $P_z < 0$,

(2) point is above view volume if $P_y > P_z$,

(3) point is below view volume if $P_y > -P_z$,

(4) point is right view volume if $P_x > P_z$,

(5) point is left view volume if $P_x > -P_z$.

In our research, a clipping procedure analogous to the polygon clipping algorithm developed by Sutherland and Hodgman [Sutherland 74] was used. Our procedure clips the polygon against each volume boundary in turn. The clipping process

produces a set of vertices defining the clipped polygon. Considering a simple case of clipping a polygon against the top clipping plane as shown in Figure A-6. The polygon is bounded by four line segments ($\overline{P_1P_2}$, $\overline{P_2P_3}$, $\overline{P_3P_4}$, and $\overline{P_4P_1}$) and its vertices P_1 and P_4 are inside and P_2 and P_3 are outside the view volume. Processing a line segment at a time, the relation between the line segment and the clip boundary is determined. Either zero, one, or two vertices are added to the output list of vertices that define the clipped polygon. There are four possible cases to be analyzed:

(1) Both vertices are below the clip plane. The line segment is acceptable, and both vertices will be added to the output list. This case is illustrated by the line segment $\overline{P_4P_1}$ in the example.

(2) One vertex is below and the other is above the clip plane. The intersecting point i is determined. The inside vertex and the intersection point are added to the output list. Examples of this case are $\overline{P_1P_2}$ and $\overline{P_3P_4}$.

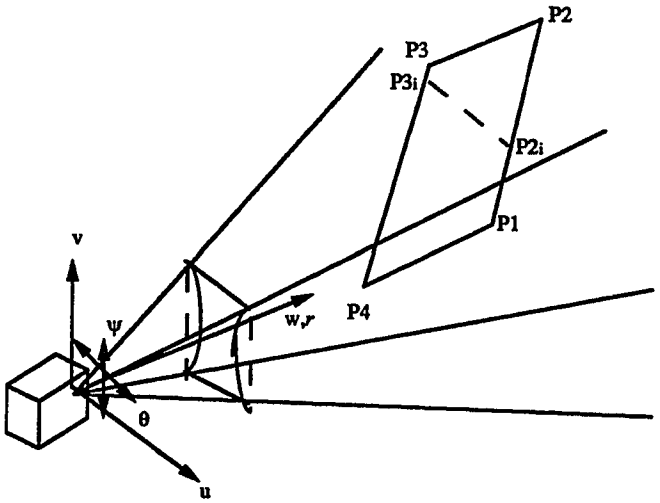


Figure A-6. Clipping against the top clip plane.

(3) One vertex is on and the other is outside the clip plane. The vertex on the clip plane will be added to the output list.

(4) Both vertices are above the clip plane. The line segmented is rejected, and no point will be added to the output list. This case is illustrated by $\overline{P_2P_3}$.

After processing each of the line segment in turn, the output list contains the vertices of the clipped polygon below the top clipping plane. The clipped polygon will then be clipped against other clipping planes. Clipping against other boundary planes occurs in much the same way. The intersecting points of a polygon edge with the boundary planes are given by

Front clipping plane:

$$i_x = \frac{- s_z(p_x - s_x)}{(p_z - s_z)}$$

$$i_y = \frac{- p_z(p_y - s_y)}{(p_z - s_z)}$$

$$i_z = 0$$

Top and bottom clipping planes:

$$i_x = \frac{(p_x - s_x)(s_z + K s_y)}{K(s_y - p_y) - (p_z - s_z)} + s_x$$

$$i_y = \frac{(p_y - s_y)(s_z + K s_y)}{K(s_y - p_y) - (p_z - s_z)} + s_y$$

$$i_z = -K i_y$$

where K = -1 for top and K = 1 for bottom clipping plane.

Right and left clipping plane:

$$i_x = \frac{(p_x - s_x)(s_z + K s_x)}{K(s_x - p_x) - (p_z - s_z)} + s_x$$

$$i_y = \frac{(p_y - s_y)(s_z + K s_x)}{K(s_x - p_x) - (p_z - s_z)} + s_y$$

$$i_z = -K i_x \qquad \text{(A.9)}$$

where K = -1 for right and K = 1 for left clipping plane.

Here *p* and *s* denote the position vector of the two vertices of the line segment.

A.4 Inverse Scaling

The scaling operation in the normalization process distorts the shape of the polygon and relative distance between points in space. The distorted shapes and distances must be restored to produce correct range data. This is done by applying an inverse scaling to the clipped polygons. That is,

$$S^{-1} = \begin{bmatrix} \cos(\Psi/2) & 0 & 0 & 0 \\ \tan(\Theta/2) & 1 & & \\ 0 & \tan(\Psi/2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}$$

$$= \begin{bmatrix} \tan(\Theta/2) & 0 & 0 & 0 \\ \cos(\Psi/2) & & & \\ 0 & \tan(\Psi/2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(A.10)}$$

A.5 Perspective projection

Once the clipped polygons are scaled, they are ready to be mapped onto the projection plane. As shown in Figure A-7, perspective projection of a point **p** onto a view plane which is d units in front of the origin is defined by,

$$p_u' = \frac{p_u}{p_w / d}$$

$$p_v' = \frac{p_v}{p_w / d}$$

$$p_w' = d \qquad \text{(A.11)}$$

where **p'** is the position vector of the image point on the view plane.

Applying the transformations defined in Equation A.11 to every vertex of the clipped polygon produces a perspective projection of the polygon onto the projection plane.

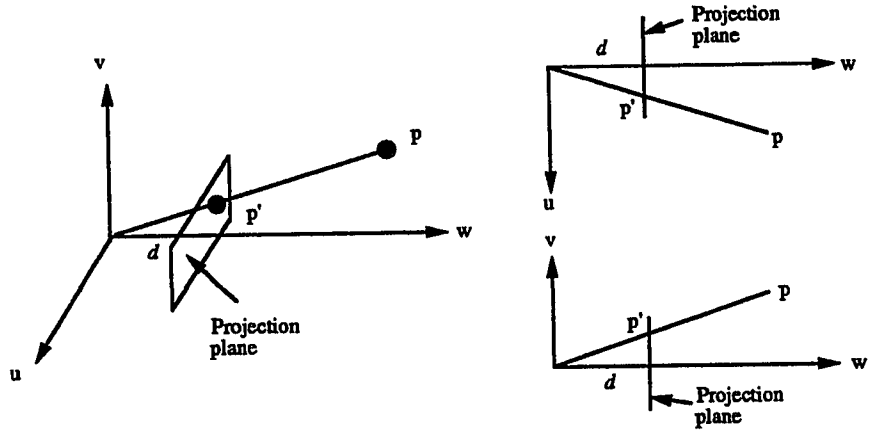


Figure A-7. Perspective projection.

Appendix B

Fitting a plane to Noisy Data by the Eigenvector Method

B.1 Fitting a Plane to Noisy Data

Plane fitting is a problem of finding the best fit plane to a set of points, $\{(x_i\ y_i\ z_i)$, i=1,2,...,m$\}$. Suppose that the plane equation of the best fit plane is given by

$$\mathbf{n}^t\mathbf{x} = d \qquad \text{(B.1)}$$

where

$\mathbf{n}^t = [a\ b\ c]$, is a 3x1 unit vector along the surface normal of the best fit plane,

$\mathbf{x}^t = [x\ y\ z]$,

$d >= 0$.

As shown in Figure B-1, the perpendicular distance of the ith point $\mathbf{x}_i$ to the plane is given by

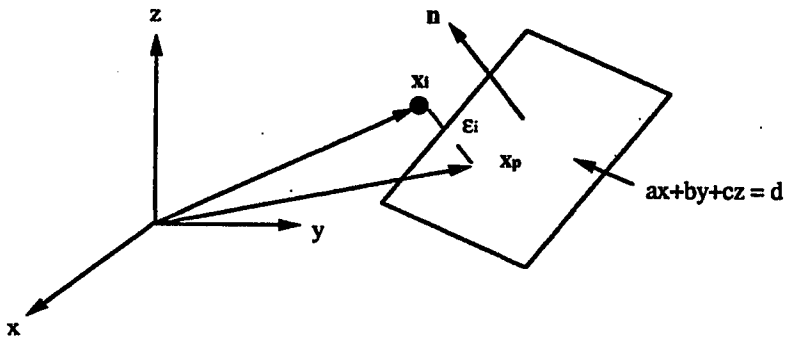


Figure B-1. Surface fitting using the eigenvector method.

$$\varepsilon_i = \mathbf{n}^t(\mathbf{x}_i - \mathbf{x}_p) \qquad \text{(B.2)}$$

where $\mathbf{x}_p$ is any point on the best fit plane.

Using the minimum square error (MSE) criterion, the best plane is that plane which minimizes the quantity

$$E = \sum_{i=1}^{m} \varepsilon_i^2 = \sum_{i=1}^{m} (\mathbf{n}^t(\mathbf{x}_i - \mathbf{x}_p))^2 = \sum_{i=1}^{m} \mathbf{n}^t(\mathbf{x}_i - \mathbf{x}_p)(\mathbf{x}_i - \mathbf{x}_p)^t\mathbf{n}$$

$$E = \mathbf{n}^t\left(\sum_{i=1}^{m}(\mathbf{x}_i - \mathbf{x}_p)(\mathbf{x}_i - \mathbf{x}_p)^t\right)\mathbf{n} = \mathbf{n}^t\mathbf{A}\mathbf{n} \qquad \text{(B.3)}$$

where

$$\mathbf{A} = \sum_{i=1}^{m}(\mathbf{x}_i - \mathbf{x}_p)(\mathbf{x}_i - \mathbf{x}_p)^t, \text{ is a 3x3 matrix.}$$

Using Equation B.3, the surface fitting can be addressed as a problem of finding the point $\mathbf{x}_p$ and the vector $\mathbf{n}^t = [a\ b\ c]$ so that the error E is minimized. $\mathbf{x}_p$ is obtained by the fact that any plane minimizing E must pass through the mean of the sample points [Duda 73, Acharya 88]. That is

$$\mathbf{x}_p = \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_i \qquad \text{(B.4)}$$

By multiplying $\mathbf{n}$ to both sides of Equation B.3, one gets

$$E\mathbf{n} = \mathbf{n}\mathbf{n}^t\mathbf{A}\mathbf{n} \qquad \text{(B.5)}$$

If $\mathbf{n}$ is an eigenvector of the symmetric matrix $\mathbf{A}$, that is

$$\mathbf{A}\mathbf{n} = \lambda\mathbf{n} \qquad \text{(B.6)}$$

then Equation B.5 can be rewritten as

$$E\mathbf{n} = \lambda\mathbf{n}\mathbf{n}^t\mathbf{n}$$

Since $\mathbf{n}$ is an unit vector, the product $\mathbf{n}^t\mathbf{n}$ equals to one. The above equation

becomes

$$E\mathbf{n} = \lambda\mathbf{n}, \text{ or}$$

$$E = \lambda. \qquad \text{(B.7)}$$

From equations B.6 and B.7, it is clear that the least square surface fitting is equivalent to an eigenvector problem: the vector $\mathbf{n}$ which minimizes E is obtained by finding the eigenvector of $\mathbf{A}$ associated with the smallest eigenvalue. Once $\mathbf{n}$ is determined, the coefficient d is given by

$$d = \mathbf{n}^t\mathbf{x}_p \qquad \text{(B.8)}$$

B.2 Probability Density Function of the Minimum Squared Error

In our research, the data points used for surface fitting are measured by a laser scanner. As shown in Figure B-2, let r denote the measured range value degraded by a known noise. That is,

$$r = r_t + r_e$$

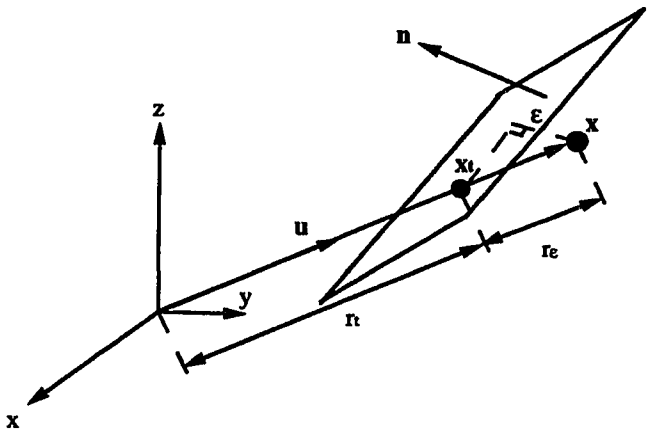


Figure B-2. Range measurement degraded by noise.

where r_t is the actual range,

$r_e = N(0,\alpha)$, is a zero-mean normally distributed random noise.

Let $\mathbf{u}$ denote the unit vector of the laser beam. Also let $\mathbf{x}_t$ denote the intersecting point of the laser with the plane (whose plane equation will be determined) and $\mathbf{x}$ denote the *measured* point. That is,

$$\mathbf{u} = \frac{\mathbf{x}}{r}$$

$$\mathbf{x} = \mathbf{x}_t + r_e\mathbf{u} \qquad \text{(B.9)}$$

Since the probability distribution function of the range noise is known, how does the surface fitting error E varies with the noise? Section B.1 has already shown that the minimum square error equals the smallest eigenvalue of the matrix A. For this reason, the question is equivalent to how the smallest eigenvalue of A varies with the noise. Substituting Equation B.9 into Equation B.2, one gets

$$\varepsilon_i = \mathbf{n}^t(\mathbf{x}_{ti} + r_e\mathbf{u}_i - \mathbf{x}_p) = \mathbf{n}^t(\mathbf{x}_{ti} - \mathbf{x}_p) + r_e\,\mathbf{n}^t\mathbf{u}_i$$
$$= r_e\,\mathbf{n}^t\mathbf{u}_i$$

or

$$\varepsilon_i = k_i\, r_e \qquad \text{(B.10)}$$

where $k_i = \mathbf{n}^t\mathbf{u}_i$, is a real number determined by the angle between the laser and surface normal of the best fit plane.

Since $r_e = N(0,\alpha)$, then $\varepsilon_i = N(0,K_i\alpha)$. From Equation B.3

$$E = \mathbf{n}^t \sum_{i=1}^{m}(\mathbf{x}_i - \mathbf{x}_p)(\mathbf{x}_i - \mathbf{x}_p)^t\,\mathbf{n}$$
$$= \mathbf{n}^t\mathbf{n}\mathbf{n}^t \sum_{i=1}^{m}(\mathbf{x}_i - \mathbf{x}_p)(\mathbf{x}_i - \mathbf{x}_p)^t\,\mathbf{n}\mathbf{n}^t\mathbf{n}$$
$$= \mathbf{n}^t \sum_{i=1}^{m}\mathbf{B}(\mathbf{x}_i - \mathbf{x}_p)(\mathbf{x}_i - \mathbf{x}_p)^t\mathbf{B}\,\mathbf{n} \qquad \text{(B.11)}$$

where $\mathbf{B} = \mathbf{nn}^t$, is a 3x3 matrix.

Using Equation B.11, E is solved by finding the smallest eigenvalue of the matrix

$$\mathbf{A}' = \sum_{i=1}^{m} \mathbf{B}(\mathbf{x}_i - \mathbf{x}_p)(\mathbf{x}_i - \mathbf{x}_p)^t\mathbf{B}$$
$$= \sum_{i=1}^{m} \mathbf{B}(\mathbf{x}_i - \mathbf{x}_p)(\mathbf{B}^t(\mathbf{x}_i - \mathbf{x}_p))^t \qquad \text{(B.12)}$$

Since

$\mathbf{B}^t = \mathbf{B}$, and from Equation B.2

$\varepsilon_i\mathbf{n} = \mathbf{nn}^t(\mathbf{x}_i - \mathbf{x}_p) = \mathbf{B}(\mathbf{x}_i - \mathbf{x}_p)$

Equation B.11 can be rewritten as

$$\mathbf{A}' = \sum_{i=1}^{m} \varepsilon_i\mathbf{n}(\varepsilon_i\mathbf{n})_t = \sum_{i=1}^{m}\varepsilon_i^2\, \mathbf{nn}^t = \sum_{i=1}^{m}\varepsilon_i^2\, \mathbf{B}$$

Since the rank of the matrix **B** is one, **A'** has only one eigenvector, (which is **n**), with a corresponding eigenvalue

$$\lambda = \sum_{i=1}^{m}\varepsilon_i^2$$

or

$$E = \sum_{i=1}^{m}\varepsilon_i^2 \qquad \text{(B.13)}$$

Notice that Equation B.13 is exactly the definition of the MSE criterion. Recall that

$\varepsilon_i = N(0,k_i\alpha)$

If K_i's calculated from the data points are close to each other, that is

$$K_i - K <= \sigma, \qquad \text{for i=1,2,...,m}$$

where

$$K = \frac{1}{m} \sum_{i=1}^{m} K_i \tag{B.14}$$

σ is a small positive number.

Then ε_i's (i=1,2,...,m) are considered as identical independent normally distributed random variables and the minimum square error E, by definition, has a chi-square distribution with m degree of freedom.

A computer simulation was performed to check the theory described above. As shown in Figure B-3, a plane 45 degrees to the laser was used in the experiments. Ten points (m = 10) were measured from the plane. A zero mean normally distributed random noise was added to the range data. The standard deviation of the random noise was 0.8% of the range values. The surface normal of the plane was determined using the minimum square error criterion described in the preceding section; the surface fitting error was recorded. Figure B-4 shows the simulation results of 1000 surface fittings and Figure B-5 shows the probability distribution function of E. The results clearly reveal that E has a chi-square distribution of 7 (m-3) degrees of freedom (rather than 10 degrees of freedom). The discrepancy in the number of the degrees of freedom derives from the initial assumption that k_i's are approximately equal. Figure B-6 shows similar results for range noise whose standard deviation equals 1% of the range measurements. Therefore, we may conclude that the probability distribution function of E is a chi-square distribution with m-3 degrees of freedom.

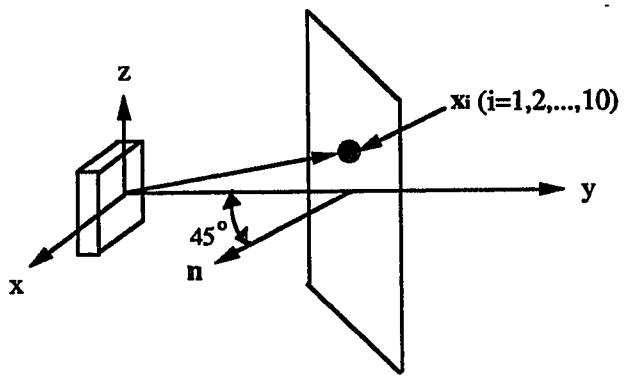


Figure B-3. Surface fitting of a plane 45 degrees to the laser.

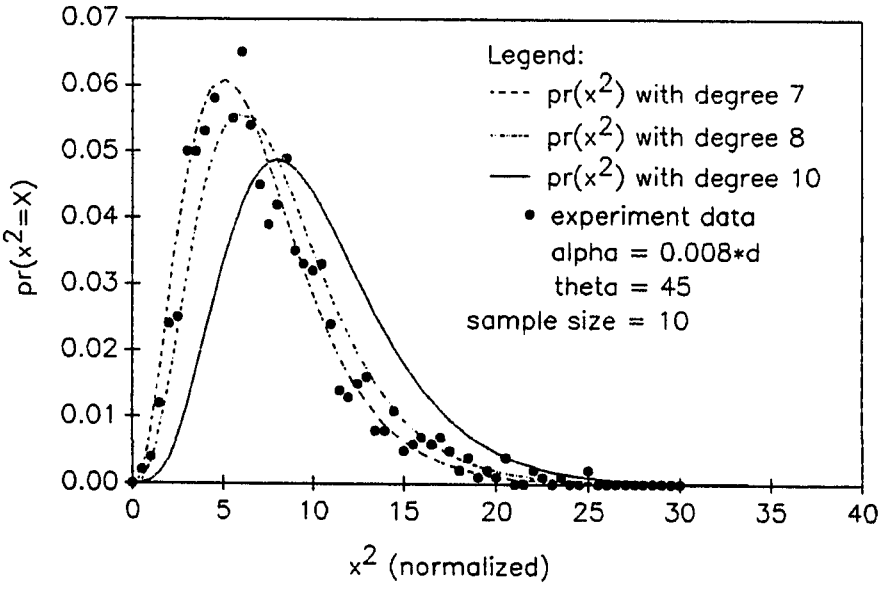


Figure B-4. Probability density function of E (based on 1000 trials).

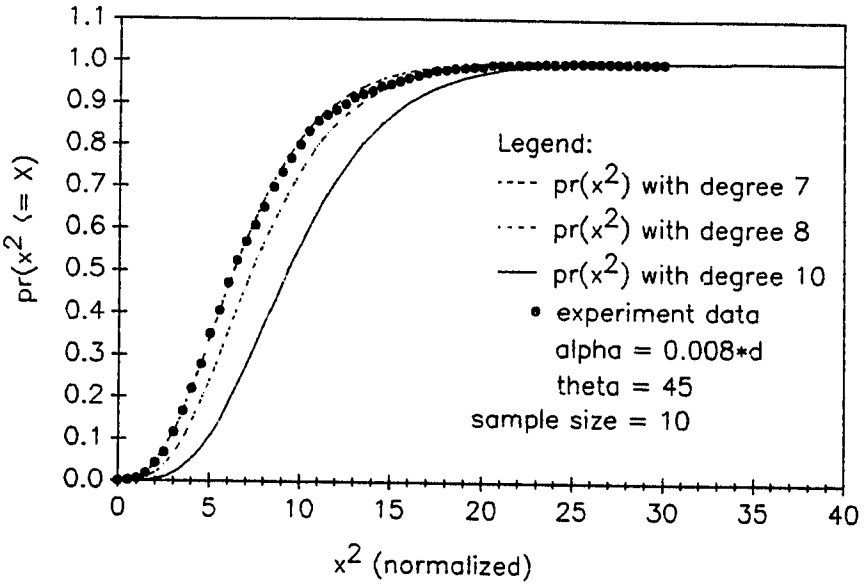


Figure B-5. Probability distribution function of E ($\alpha = 0.008r$).

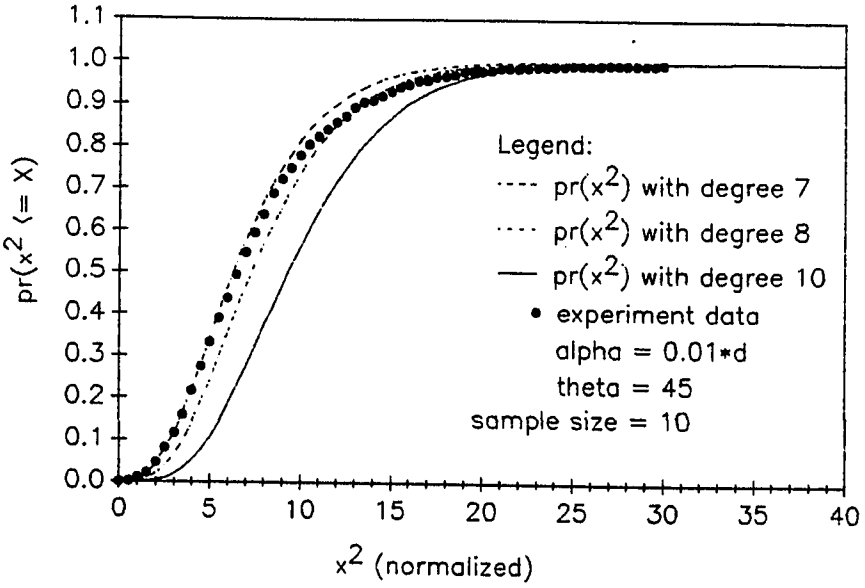


Figure B-6. Probability distribution function of E ($\alpha = 0.01r$).

Appendix C

Computer Program Listing

This section includes the software implementations of the laser imaging simulation program (LISA), the noise insensitive edge detector, and the sensor-based direct search algorithm (SDSA). All of the programs were implemented in C language on an IBM-AT. A Microsoft C compiler was used to compile the program. Table C-1 provides a summary of the programs.

Table C-1 Summary of the programs

Name	Description	Page
lasimage.c	laser imaging simulation program	144
get.c	data entry subroutine	154
clip_3d.c	3-D clipping subroutine	165
set_3d_r.c	3-D rotation subroutine	179
drawline.c	line drawing subroutine	181
rand_nor.c	Gausian random number generator	182
norm_per.c	view volume normalization subroutine	183
shad.c	range and reflectance data generation subroutine	185
main_sha.c	dummy main program for calling shad.c	203
spath.c	minimum distance collision-free path planner	206
navigate.c	path searching subroutine	224
simplex.c	downhill Nelder-Mead simplex searching algorithm	230
buildmap.c	elevation map plotting subroutine	234
edge.c	noise insensitive edge detector	247
mateig.c	calculate eigen values and corresponding eigen vectors of a real symmetric	255

```
/* LISA - Laser Imaging Simulation Algorithm

        This program simulates scanning laser range finders. It generates laser range
        and reflectance data. The system parameters of the simulated laser range
        finders are input interactively.

        Compiled with Microsoft C compiler (version 5.0 or later) and C_tool plus
        utility library.

        by Chris K. Wu, Nov. 1988
*/

#include <variable.h>
#include <bscreen.h>
#include <bgraph.h>
#include <bkeybd.h>
#include <bapplic.h>
#include <bdirect.h>
#include <malloc.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <process.h>

/*#define EXEC_SHAD*/

char            string[81];
int             h_pixel,v_pixel;
int             las_i_pixel,las_j_pixel;
int             amb_int;
int             I_CENTER,J_CENTER,I_PIXEL,J_PIXEL,res_mode;
int             cindex[MAXVERT],cpindex[MAXPOLY];
int             npvert[MAXPOLY];
int             indxlist[MAXVERT];
int             nvert, npoly;
int             num_clip_ver;
short           *pvertptr[MAXPOLY];
short           pvert[TOTALPVERT];
float           *wldx, *wldy;
float           *ndcx, *ndcy;
float           rot_3d[4][4];
float           nper[4][4];
float           view_plane_i[3],view_plane_j[3],view_plane_k[3];
float           bbox[3][2];
float           vertices[MAXVERT][4];
float           lx, nlx, ly, nly, lz, nlz;
```

```
float           vrp[3],cop[3];
float           lgt[3],r_angle;
float           pitch,yaw;
float           h_view_angle,v_view_angle;
float           h_scan_angle,v_scan_angle;
float           delta_y;
float           xlist[MAXVERT], ylist[MAXVERT], zlist[MAXVERT];
float           dxlist[MAXVERT],dylist[MAXVERT],dzlist[MAXVERT];
float           planeq[4];
float           clip_ver[MAXPVERT*2][3];

#ifdef EXEC_SHAD
float huge      *z_buf;
float huge      *r_buf;
struct et_slot  et_entry[MAXPVERT*2],*et[MAX_Y_SCAN];
struct h_edge   het_entry[MAXPVERT/2],*het[MAX_Y_SCAN];
#endif

main()
{

char            data_file[20],msg[80],s_yes[4],ch,ans1[5],ans2[5];
short           iindex;
int             row,left_margin,j,k,laser_image,f_status;
int             v1,v2,v3,vtmp;
int             key,num_scroll,loop_flag;
int             preview_mode,add_noise;
long            i,array_size;
long            h_return,ltemp;
float           *left_vec,right_vec[4];
float           x,y,z,length;
float           plane_ver[MAXPVERT][4];
float           plane_norm[MAXPVERT][3];
float           a[2],b[2];
float           lambda,ar,ft,t,snr,lambdam,noise;
FILE            *fopen(),*fp;
FSPEC           pfile;

sprintf(s_yes,"yes");
loop_flag = 0;
program_loop:
sprintf(msg,"Laser Imaging Simulation Program");
title_box(msg);
row=5;
left_margin=5;
```

```
/*
Retrieve data from the specified file
*/
sprintf(msg,"lasimage.dat");
f_status = 0;
if ( (drsfirst(msg,0,&pfile)) == 0 ) {
        fp = fopen(msg,"r");
        fscanf(fp,"%s",data_file);
        fscanf(fp,"%d",&preview_mode);
        fscanf(fp,"%s",ans1);
        fscanf(fp,"%f %f %f %f %f",&lambda,&ar,&ft,&t,&lambdam);
        fscanf(fp,"%s",ans2);
        fclose(fp);
        }
else {
        sprintf(data_file,"demo.dat");
        preview_mode = 1;
        sprintf(ans1,"no");
        lambda = 632.8;
        ar = 1.5e-4;
        ft = 5;
        t = 0.5;
        lambdam = 5.;
        sprintf(ans2,"no");
        f_status = 1;
        }
sprintf(msg,"Enter the name of the data file [default %s] ?",data_file);
scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
sccurset(row++,left_margin+61);
msg[0]= '\0';
apquery(msg,sizeof(msg),&key,&num_scroll);
if (strlen(msg) != 0) {
        strcpy(data_file,msg);
        loop_flag = 0;
        f_status = 1;
        }
if (!loop_flag)
        if (getobj(data_file))
                exit(1);
sprintf(msg,"select the operation mode");
scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
sprintf(msg,"(1) wireframe");
scwrstr(row++,left_margin+5,0,msg,-1,-1,WRTMODE);
sprintf(msg,"(2) wireframe + range");
scwrstr(row++,left_margin+5,0,msg,-1,-1,WRTMODE);
sprintf(msg,"your choice [default %1d] ? ",preview_mode);
```

```
scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
sccurset(row++,left_margin+26);
msg[0]= '\0';
apquery(msg,sizeof(msg),&key,&num_scroll);
if (strlen(msg) != 0) {
        preview_mode = atoi(msg);
        f_status = 1;
        }
if (preview_mode == 2) {
        laser_image = 1;
        sprintf(msg,"Do you want to add noise to the laser image [default %s] ? "
                        ,ans1);
        scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
        sccurset(row++,left_margin+60);
        msg[0]= '\0';
        apquery(msg,sizeof(msg),&key,&num_scroll);
        if (strlen(msg) != 0) {
                strcpy(ans1,msg);
                f_status = 1;
                }
        if ( strncmp(ans1,s_yes,1))
                add_noise = 0;
        else
                add_noise = 1;
        if (add_noise) {
                sprintf(msg,"Enter your instrument parameters :");
                scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
                left_margin = 10;
                sprintf(msg,"wavelength of the laser beam (nM) [default %5.1f] ?",
                                lambda);
                scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
                sccurset(row++,left_margin+52);
                msg[0]= '\0';
                apquery(msg,sizeof(msg),&key,&num_scroll);
                if (strlen(msg) != 0) {
                        lambda = atof(msg);
                        f_status = 1;
                        }
                sprintf(msg,"power of the laser beam (mW) [default %4.1f] ?",ft);
                scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
                sccurset(row++,left_margin+46);
                msg[0]= '\0';
                apquery(msg,sizeof(msg),&key,&num_scroll);
                if (strlen(msg) != 0) {
                        ft = atof(msg);
                        f_status = 1;
```

```
                }
            sprintf(msg,"modulation frequency (MHz) [default %5.1f] ?",
                    lambdam);
            scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
            sccurset(row++,left_margin+45);
            msg[0]= '\0';
            apquery(msg,sizeof(msg),&key,&num_scroll);
            if (strlen(msg) != 0) {
                lambdam = atof(msg);
                f_status = 1;
                }
            sprintf(msg,
                "capture area of the receiver (M**2) [default %7.2e] ?",ar);
            scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
            sccurset(row++,left_margin+58);
            msg[0]= '\0';
            apquery(msg,sizeof(msg),&key,&num_scroll);
            if (strlen(msg) != 0) {
                ar = atof(msg);
                f_status = 1;
                }
            sprintf(msg,"acquisition time per frame (sec) [default %4.2f] ?",t);
            scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
            sccurset(row++,left_margin+50);
            msg[0]= '\0';
            apquery(msg,sizeof(msg),&key,&num_scroll);
            if (strlen(msg) != 0) {
                t = atof(msg);
                f_status = 1;
                }
            snr = 0.25*0.1*lambda*1e-9*ar*ft*1e-3*t*0.3*0.7071;
            snr /= (3.14159*1.9878e-25*25*16384);
            snr = sqrt(snr);
            noise = 3.e2/(8.8858*snr*lambdam*5);
            left_margin = 5;
            sprintf(msg,"the signal-noise-ratio of your system is approximatly");
            scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
            sprintf(msg,"%5.1f dB at a distance of 5 meters (alpha = %7.4f).",
            log10(snr)*20,noise);
            scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
            }
        }
else
        laser_image = 0;
```

```
sprintf(msg,
        "Is this computer equiped with an EGA display [default %s] ? ",ans2);
scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
sccurset(row++,left_margin+60);
msg[0]= '\0';
apquery(msg,sizeof(msg),&key,&num_scroll);
if (strlen(msg) != 0) {
        strcpy(ans2,msg);
        f_status = 1;
        }
if ( strncmp(ans2,s_yes,1)) {
        res_mode = 6;
        I_CENTER = 380;
        J_CENTER = 100;
        I_PIXEL = 217;
        J_PIXEL = 99;
        }
else {
        res_mode = 16;
        I_CENTER = 380;
        J_CENTER = 175;
        I_PIXEL = 217;
        J_PIXEL = 174;
        }
if (f_status) {
        sprintf(msg,"lasimage.dat");
        fp = fopen(msg,"w");
        fprintf(fp,"%s\n",data_file);
        fprintf(fp,"%d\n",preview_mode);
        fprintf(fp,"%s\n",ans1);
        fprintf(fp,"%f %7.2e %f %f %f\n",lambda,ar,ft,t,lambdam);
        fprintf(fp,"%s\n",ans2);
        fclose(fp);
        }

/*
aquire the viewing parameters
*/
getxyz(preview_mode);
/*
set up the normalization matrix nper
*/
set_up_nper(preview_mode);
```

```
/*
acquire memory space for z_buffer and refresh buffer
*/
#ifdef EXEC_SHAD
if (laser_image) {
        array_size = (long)(h_pixel + 1 ) * (long)(v_pixel + 1);
        z_buf = (float huge *)halloc(array_size,sizeof(float));
        if (z_buf == NULL) {
                printf("Insufficient memory available\n");
                exit(1);
                }
        r_buf = (float huge *)halloc(array_size,sizeof(float));
        if (r_buf == NULL) {
                printf("Insufficient memory available\n");
                exit(1);
                }
        for (i=0;i<array_size;i++)
                *(z_buf+i) = 10E20;
        }

#endif

/*
save data for child process
*/
if (laser_image) {
        fp = fopen("child.dat","w");
        fprintf(fp,"%d %d\n",h_pixel,v_pixel);
        fprintf(fp,"%f %f\n",h_scan_angle,v_scan_angle);
        fprintf(fp,"%d\n",amb_int);
        fprintf(fp,"%d\n",npoly);
        fprintf(fp,"%d\n",add_noise);
        fprintf(fp,"%f %7.2e %f %f %f\n",lambda,ar,ft,t,lambdam);
        }

grinit(res_mode,0,0);
printf("data file:\n");
printf("%s\n",data_file);
printf("\n");
printf("operation mode:\n");
if (preview_mode == 1)
        printf("wireframe\n");
else
        printf("wireframe + range\n");
printf("\n");
if (preview_mode == 1) {
```

```
        printf("field of view:\n");
        printf("%5.1f x %5.1f degrees\n",v_view_angle,h_view_angle);
        }
else {
        printf("scan angles:\n");
        printf("%5.1f x %5.1f degrees\n",v_scan_angle,h_scan_angle);
        }
printf("\n");
printf("viewer position\n");
printf("x = %7.2f\n",cop[0]);
printf("y = %7.2f\n",cop[1]);
printf("z = %7.2f\n",cop[2]);
printf("\n");
printf("viewer orientation:\n");
printf("pitch = %7.2f\n",pitch);
printf("yaw   = %7.2f\n",yaw);
printf("roll  = %7.2f\n",r_angle);
printf("        (degrees)\n");
printf("\n");
printf("processing ...\n");
/*
clip each surfaces against the canonical view volumn
*/
for (i=0;i<npoly;i++) {

/*
read vertex data of plane i
*/
        for (j=0;j<npvert[i];j++) {
                iindex = *(pvertptr[i]+j);
                for (k=0;k<4;k++)
                        plane_ver[j][k]=vertices[iindex][k];
                }

/*
transform the view volumn to the canonical view volumn
*/

        for (j=0;j<npvert[i];j++) {
                left_vec = &(plane_ver[j][0]);
                vdotm(left_vec,nper,4,4,right_vec);
                for (k=0;k<3;k++)
                        plane_norm[j][k] = right_vec[k];
                }
```

```
/*
clip against the canonical view volum
*/

        clip_3d(plane_norm,npvert[i],clip_ver,&num_clip_ver);

/*
save data for child process
*/
if (laser_image) {
        fprintf(fp,"%d\n",num_clip_ver);
                for (j=0; j<num_clip_ver; j++)
                        fprintf(fp,"%f %f %f\n",clip_ver[j][0],clip_ver[j][1],
                                clip_ver[j][2]);
        }
/*
draw wireframe representation of the clipped surface
*/
        if (num_clip_ver > 0) {
                for (j=0; j<num_clip_ver-1; j++) {
                        a[0] = clip_ver[j][0] / clip_ver[j][2];
                        a[1] = clip_ver[j][1] / clip_ver[j][2];
                        b[0] = clip_ver[j+1][0] / clip_ver[j+1][2];
                        b[1] = clip_ver[j+1][1] / clip_ver[j+1][2];
                        drawline(a,b);
                        }
                a[0] = clip_ver[num_clip_ver-1][0] / clip_ver[num_clip_ver-1][2];
                a[1] = clip_ver[num_clip_ver-1][1] / clip_ver[num_clip_ver-1][2];
                b[0] = clip_ver[0][0] / clip_ver[0][2];
                b[1] = clip_ver[0][1] / clip_ver[0][2];
                drawline(a,b);
                }
        }

/*
generate the laser image
*/

if (laser_image) {
        fclose(fp);
        sprintf(msg,"main_sha.exe");
        system(msg);
        }
printf("Done");
kbflush();
```

```
do { } while ( 0 == kbready(&ch,&key) );
kbin(&key);

#ifdef EXEC_SHAD
if (laser_image) {
        hfree((char huge *)r_buf);
        hfree((char huge *)z_buf);
        }
#endif

grinit(0,0,0);
sprintf(msg,"Laser Imaging Simulation Program");
title_box(msg);
row=5;
left_margin=20;
sprintf(msg,"continue [default yes] ? ");
sprintf(ans2,"yes");
scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
sccurset(row++,left_margin+25);
msg[0]= '\0';
apquery(msg,sizeof(msg),&key,&num_scroll);
if (strlen(msg) != 0)
        strcpy(ans2,msg);
if ( !strncmp(ans2,s_yes,1)) {
        loop_flag = 1;
        goto program_loop;
        }
scpclr();
exit(0);
}
```

```
/*
getobj.c --
        This subroutine reads the data file of the 3-D objects. The 3-D objects are
        modeled using the surface boundary representation method. Plane equations
        of each surfaces are also calculated.

ARGUMENTS:

        bbox[3][2]    :     rectangular box which encloses the objects,
        vertices[i][4] :    position vector of the ith vertex,
        MAXPVERT :          maximum total polygon vertices,
        npvert[i]     :     number of vertices of polygon i,
        pvert[]       :     topology array contains vertices of each polygon,
        pvertptr[i]   :     index vector for retrieving vertex sequence for polygon
                            i from pvert[],
 */
#include      <stdio.h>
#include      <math.h>
#include      <string.h>
#include      <bapplic.h>
#include      <bscreen.h>
#include      <bdirect.h>
#include      "variable.h"
#define       X_DEFAULT       0
#define       Y_DEFAULT       6
#define       Z_DEFAULT       10
#define       H_VIEW          45
#define       V_VIEW          45
#define       PITCH           -60
#define       YAW             0
#define       ROLL            0
#define       H_ANGLE         80
#define       V_ANGLE         30
#define       H_PIXEL         128
#define       V_PIXEL         128

int getobj(filename)
char *filename;
{
extern float    bbox[3][2];
extern float    vertices[MAXVERT][4];
extern int      npvert[MAXPOLY];
extern short    *pvertptr[MAXPOLY];
extern short    pvert[TOTALPVERT];
extern int      nvert, npoly;
int             i,j,k;
```

```
short           vtmp,v1,v2,v3;
float           ftmp,maxd,scale,offset[3];
float           x,y,z,x0,y0,z0,length;
FILE            *fp,*fopen();

if ((fp=fopen(filename,"r")) == NULL) {
        printf("Can't open file: %s\n",filename);
        return(1);
        }
fscanf(fp,"%d%d",&nvert,&npoly);
if ((nvert > MAXVERT) || (npoly > MAXPOLY)) {
        printf("Object has vertices greater than %3d or\n",MAXVERT);
        printf("number of polygons greater than %3d.\n",MAXPOLY);
        return(2);
        }
fscanf(fp,"%f%f%f%f%f%f",&bbox[0][0],&bbox[0][1],&bbox[1][0],
        &bbox[1][1],&bbox[2][0],&bbox[2][1]);
maxd=0.0;

/*
Read vertices from data file and perform translation and scaling operations
*/
for (i=0;i<nvert;i++) {
        fscanf(fp,"%f%f%f",&vertices[i][0],&vertices[i][1],
                &vertices[i][2]);
        vertices[i][3] = 1.0;
        }

/*
Read polygon vertices
*/
k=0;
for (i=0;i<npoly;i++) {
        fscanf(fp,"%d",&npvert[i]);
        if ( npvert[i] > MAXPVERT) {
                printf("Polygon %3d has vertices greater than %3d",i,MAXPVERT);
                return(3);
                }
        if ((k+npvert[i]) > TOTALPVERT) {
                printf("Total polygon vertices greater than %4d\n",TOTALPVERT);
                return(4);
                }
        pvertptr[i] = &pvert[k];
```

```
        for(j=0;j<npvert[i];j++) {
                fscanf(fp,"%hd",&vtmp);
                pvert[k++] = vtmp -1;
                }
        }
fclose(fp);
return(0);
}

/*
getxyz() --
requests input of
        (1) view reference point position,
        (2) light source location,
        (3) view plane normal,
        (4) horizontal scan angle
        (5) horizontal pixel resolution
        (6) vertical scan angle
        (7) vertical pixel resolution
        (8) ambiguity interval
        (9) roll angle of the laser scanner about its optical axis.
It also calculates
        (1) center of projection which is one unit length above the
            view reference point along the view plane normal
        (2) axes of the left-handed view plane coordinates.
*/
getxyz(preview_mode)
int     preview_mode;
{
extern float    vrp[3],cop[3];
extern float    lgt[3],r_angle;
extern float    h_view_angle,v_view_angle;
extern float    h_scan_angle,v_scan_angle;
extern float    pitch,yaw;
extern int      h_pixel,v_pixel;
extern int      las_i_pixel,las_j_pixel;
extern int      amb_int;
extern float    rot_3d[4][4];
extern float    view_plane_i[3],view_plane_j[3],view_plane_k[3];
char            msg[80],in_buf[80];
static char     str1[] = "x= ";
static char     str2[] = "y= ";
static char     str3[] = "z= ";
int             i,right_fill,row,left_margin,key,num_scroll,f_status;
float           atemp[4],result[4],vup[3];
float           vpn[3],theta,length;
```

```
FSPEC       pfile;
FILE        *fopen(),*fp;

sprintf(msg,"Laser Imaging Simulation Program");
title_box(msg);
row=3;
left_margin=10;

/*
check if previously defined parameters are available
*/
sprintf(msg,"getxyz.dat");
f_status = 0;
if ( (drsfirst(msg,0,&pfile)) == 0 ) {
        fp = fopen(msg,"r");
        fscanf(fp,"%f %f %f",&cop[0],&cop[1],&cop[2]);
        fscanf(fp,"%f %f %f",&lgt[0],&lgt[1],&lgt[2]);
        fscanf(fp,"%f %f %f",&pitch,&yaw,&r_angle);
        fscanf(fp,"%f %f",&h_view_angle,&v_view_angle);
        fscanf(fp,"%f %d",&h_scan_angle,&h_pixel);
        fscanf(fp,"%f %d",&v_scan_angle,&v_pixel);
        fscanf(fp,"%d",&amb_int);
        fclose(fp);
        }
else {
        cop[0] = X_DEFAULT;
        cop[1] = Y_DEFAULT;
        cop[2] = Z_DEFAULT;
        lgt[0] = X_DEFAULT;
        lgt[1] = Y_DEFAULT;
        lgt[2] = Z_DEFAULT;
        pitch  = PITCH;
        yaw    = YAW;
        r_angle   = ROLL;
        h_view_angle = H_VIEW;
        v_view_angle = V_VIEW;
        h_scan_angle = H_ANGLE;
        h_pixel = H_PIXEL;
        v_scan_angle = V_ANGLE;
        v_pixel = V_PIXEL;
        amb_int = 32;
        f_status = 1;
        }
```

```
/*
Enter the viewer position
*/
sprintf(msg,
        "Enter the viewer location [default :%5.1f,%5.1f,%5.1f]",
                                        cop[0],cop[1],cop[2]);
scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
left_margin=15;
scwrstr(row,left_margin,0,str1,-1,-1,WRTMODE);
sccurset(row++,left_margin+3);
in_buf[0]= '\0';
apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
if (strlen(in_buf) != 0) {
        cop[0] = atof(in_buf);
        f_status = 1;
        }
scwrstr(row,left_margin,0,str2,-1,-1,WRTMODE);
sccurset(row++,left_margin+3);
in_buf[0]= '\0';
apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
if (strlen(in_buf) != 0) {
        cop[1] = atof(in_buf);
        f_status = 1;
        }
scwrstr(row,left_margin,0,str3,-1,-1,WRTMODE);
sccurset(row++,left_margin+3);
in_buf[0]= '\0';
apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
if (strlen(in_buf) != 0) {
        cop[2] = atof(in_buf);
        f_status = 1;
        }

/*
Enter the light source location. Default to the viewer position.
*/
sprintf(msg,"Enter the light source location [default :%5.1f,%5.1f,%5.1f]",
                                        lgt[0],lgt[1],lgt[2]);
left_margin=10;
scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
left_margin=15;
scwrstr(row,left_margin,0,str1,-1,-1,WRTMODE);
sccurset(row++,left_margin+3);
in_buf[0]= '\0';
apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
```

```
if (strlen(in_buf) != 0) {
        lgt[0] = atof(in_buf);
        f_status = 1;
        }
scwrstr(row,left_margin,0,str2,-1,-1,WRTMODE);
sccurset(row++,left_margin+3);
in_buf[0]= '\0';
apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
if (strlen(in_buf) != 0) {
        lgt[1] = atof(in_buf);
        f_status = 1;
        }
scwrstr(row,left_margin,0,str3,-1,-1,WRTMODE);
sccurset(row++,left_margin+3);
in_buf[0]= '\0';
apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
if (strlen(in_buf) != 0)
        lgt[2] = atof(in_buf);

/*
Enter the viewer orientation.
*/
sprintf(msg,"Enter the viewer orientation [default :%5.1f,%5.1f,%5.1f degrees]",
                pitch,yaw,r_angle);
left_margin=10;
scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
sprintf(msg,"pitch =");
left_margin=15;
scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
sccurset(row++,left_margin+9);
in_buf[0]= '\0';
apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
if (strlen(in_buf) != 0) {
        pitch = atof(in_buf);
        f_status = 1;
        }
sprintf(msg,"yaw =");
scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
sccurset(row++,left_margin+7);
in_buf[0]= '\0';
apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
if (strlen(in_buf) != 0) {
        yaw = atof(in_buf);
        f_status = 1;
        }
```

```
sprintf(msg,"roll =");
scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
sccurset(row++,left_margin+8);
in_buf[0]= '\0';
apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
if (strlen(in_buf) != 0) {
        r_angle = atof(in_buf);
        f_status = 1;
        }

theta = pitch*3.14159/180;
vpn[2] = sin(theta);
length = cos(theta);
theta = yaw*3.14159/180;
vpn[0] = length * cos(theta);
vpn[1] = length * sin(theta);

sprintf(msg,"Laser Imaging Simulation Program");
title_box(msg);
row=3;
left_margin=5;

/*
Enter the horizontal view angle
*/
if (preview_mode == 1) {
        sprintf(msg,"Enter the horizontal view angle [default : %5.1f degrees] ? ",
                                        h_view_angle);
        scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
        sccurset(row++,left_margin+60);
        in_buf[0]= '\0';
        apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
        if (strlen(in_buf) != 0) {
                h_view_angle = atof(in_buf);
                f_status = 1;
                }

/*
Enter the vertical view angle
*/
        sprintf(msg,"Enter the vertical view angle [default : %5.1f degrees] ? ",
                                        v_view_angle);
        scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
        sccurset(row++,left_margin+58);
        in_buf[0]= '\0';
        apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
```

```
        if (strlen(in_buf) != 0) {
                v_view_angle = atof(in_buf);
                f_status = 1;
                }
        }
else {

/*
Enter the horizontal scanning angle
*/
        sprintf(msg,"Enter the horizontal scanning angle of the laser beem");
        scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
        sprintf(msg,"[default : %5.1f degrees]",h_scan_angle);
        scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
        left_margin=20;
        sprintf(msg,"theta= ");
        scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
        sccurset(row++,left_margin+7);
        in_buf[0]= '\0';
        apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
        if (strlen(in_buf) != 0) {
                h_scan_angle = atof(in_buf);
                f_status = 1;
                }

/*
Enter the number of pixels along the horizontal scanning line
*/
        sprintf(msg,"Enter the number of pixels along the horizontal scanning line");
        left_margin=5;
        scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
        sprintf(msg,"[default : %4d pixels]",h_pixel);
        scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
        left_margin=20;
        sprintf(msg,"number= ");
        scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
        sccurset(row++,left_margin+8);
        in_buf[0]= '\0';
        apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
        if (strlen(in_buf) != 0) {
                h_pixel = atof(in_buf);
                f_status = 1;
                }
        las_i_pixel = h_pixel / 2;
```

```
/*
Enter the vertical scanning angle
*/
        sprintf(msg,"Enter the vertical scanning angle of the laser beam");
        left_margin=5;
        scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
        sprintf(msg,"[default : %5.1f degrees]",v_scan_angle);
        scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
        left_margin=20;
        sprintf(msg,"theta= ");
        scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
        sccurset(row++,left_margin+7);
        in_buf[0]= '\0';
        apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
        if (strlen(in_buf) != 0) {
                v_scan_angle = atof(in_buf);
                f_status = 1;
        }

/*
Enter the number of pixels along the vertical scanning line
*/
        sprintf(msg,"Enter the number of pixels along the vertical scanning line");
        left_margin=5;
        scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
        sprintf(msg,"[default : %4d pixels]",v_pixel);
        scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
        left_margin=20;
        sprintf(msg,"number= ");
        scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
        sccurset(row++,left_margin+8);
        in_buf[0]= '\0';
        apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
        if (strlen(in_buf) != 0) {
                v_pixel = atof(in_buf);
                f_status = 1;
                }
        las_j_pixel = v_pixel / 2;

/*
Enter the ambiguity interval
*/
        sprintf(msg,"Enter the ambiguity interval [default : %d]",amb_int);
        left_margin=5;
        scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
        left_margin=20;
```

```
        sprintf(msg,"interval= ");
        scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
        sccurset(row++,left_margin+10);
        in_buf[0]= '\0';
        apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
        if (strlen(in_buf) != 0) {
                amb_int = atoi(in_buf);
                f_status = 1;
                }
        }
/*
Enter the imager orientation


sprintf(msg,
                "Enter the roll angle of the range finder [default : %5.1f degrees]",
                r_angle);
left_margin=5;
scwrstr(row++,left_margin,0,msg,-1,-1,WRTMODE);
left_margin=20;
sprintf(msg,"theta= ");
scwrstr(row,left_margin,0,msg,-1,-1,WRTMODE);
sccurset(row++,left_margin+7);
in_buf[0]= '\0';
apquery(in_buf,sizeof(in_buf),&key,&num_scroll);
if (strlen(in_buf) != 0) {
        r_angle = atof(in_buf);
        f_status = 1;
        }*/

if (f_status) {
        sprintf(msg,"getxyz.dat");
        fp = fopen(msg,"w");
        fprintf(fp,"%f %f %f\n",cop[0],cop[1],cop[2]);
        fprintf(fp,"%f %f %f\n",lgt[0],lgt[1],lgt[2]);
        fprintf(fp,"%f %f %f\n",pitch,yaw,r_angle);
        fprintf(fp,"%f %f\n",h_view_angle,v_view_angle);
        fprintf(fp,"%f %d\n",h_scan_angle,h_pixel);
        fprintf(fp,"%f %d\n",v_scan_angle,v_pixel);
        fprintf(fp,"%d\n",amb_int);
/*      fprintf(fp,"%f\n",r_angle);*/
        fclose(fp);
        }
/*norm_vec_3(vpn)*/
```

```
/*
Calculate the view reference point which is 1 unit
length below the viewer location along the view axis.
*/
vrp[0] = cop[0] + vpn[0];
vrp[1] = cop[1] + vpn[1];
vrp[2] = cop[2] + vpn[2];

/*
Calculate the view-up vector
*/
if((vpn[0] == 0) && (vpn[1] == 0)) {
        vup[0]=vup[2]=0.0;
        vup[1]=vpn[2];
        }
else    {
        vup[0]=vup[1]=0;
        vup[2]=1;
        }
for (i=0;i<3;i++)       view_plane_k[i] = vpn[i];
vcrossv(view_plane_k,vup,view_plane_i);
norm_vec_3(view_plane_i);
vcrossv(view_plane_i,view_plane_k,view_plane_j);
for (i=0;i<3;i++)
        atemp[i]=0;
set_3d_rot_matrix(atemp,vpn,r_angle);
for (i=0;i<3;i++)
        atemp[i]=view_plane_i[i];
atemp[3]=1;
vdotm(atemp,rot_3d,4,4,result);
for (i=0;i<3;i++)
        view_plane_i[i]=result[i];
for (i=0;i<3;i++)
        atemp[i]=view_plane_j[i];
vdotm(atemp,rot_3d,4,4,result);
for (i=0;i<3;i++)
        view_plane_j[i]=result[i];
scpclr();
}
```

```
/*
clip_3d.c --
        This subroutine clips the input plane against the 3-D canonical
        view volumn.
*/

#include "variable.h"

clip_3d(in_ver,num_in_ver,out_ver,num_out_ver)
float           *in_ver,*out_ver;
int             num_in_ver,*num_out_ver;
{
float           *p1,*p2,pv[3],*tmp_ptr;
float           old_ver[MAXPVERT*2][3],new_ver[MAXPVERT*2][3];
int             i,j,k,switch_code,num_old_ver,num_new_ver;

*num_out_ver = 0;

/*
clip against the top plane of the canonical view volumn
*/
if (num_in_ver > 1) {
num_old_ver=0;
p1 = in_ver;
p2 = in_ver + 3;
clip_top(p1,p2,pv,&switch_code);
switch (switch_code) {
        case 1: /* pt1 is outside the view volumn and pt2 is inside */
                for (j=0;j<3;j++)
                        old_ver[num_old_ver][j] = pv[j];
                num_old_ver++;
                if (!((pv[0] == *p2)&&(pv[1] == *(p2+1))&&(pv[2] == *(p2+2)))) {
                        for (j=0;j<3;j++)
                                old_ver[num_old_ver][j] = *(p2 + j);
                        num_old_ver++;
                        }
                break;
        case 2: /* pt1 is inside the view volumn and pt2 is outside */
                for (j=0;j<3;j++)
                        old_ver[num_old_ver][j] = *(p1 + j);
                num_old_ver++;
                if (!((pv[0] == *p1)&&(pv[1] == *(p1+1))&&(pv[2] == *(p1+2)))) {
                        for (j=0;j<3;j++)
                                old_ver[num_old_ver][j] = pv[j];
                        num_old_ver++;
                        }
```

```
                break;
        case 3:
                for (j=0;j<3;j++)
                        old_ver[num_old_ver][j] = *(p1 + j);
                num_old_ver++;
                for (j=0;j<3;j++)
                        old_ver[num_old_ver][j] = *(p2 + j);
                num_old_ver++;
        }
for (i=1;i<num_in_ver-1;i++) {
        p1 = in_ver + i * 3;
        p2 = in_ver + (i+1) * 3;
        clip_top(p1,p2,pv,&switch_code);
        switch (switch_code) {
                case 1:
                        for (j=0;j<3;j++)
                                old_ver[num_old_ver][j] = pv[j];
                        num_old_ver++;
                        if (!((pv[0] == *p2) && (pv[1] == *(p2+1)) && (pv[2] ==
                                *(p2+2)))) {
                                for (j=0;j<3;j++)
                                        old_ver[num_old_ver][j] = *(p2 + j);
                                num_old_ver++;
                                }
                        break;
                case 2:
                        if (!((pv[0] == *p1) && (pv[1] == *(p1+1)) && (pv[2] ==
                                *(p1+2)))) {
                                for (j=0;j<3;j++)
                                        old_ver[num_old_ver][j] = pv[j];
                                num_old_ver++;
                                }
                        break;
                case 3:
                        for (j=0;j<3;j++)
                                old_ver[num_old_ver][j] = *(p2 + j);
                        num_old_ver++;
                }
        }
p1 = in_ver + 3*(num_in_ver-1);
p2 = in_ver;
clip_top(p1,p2,pv,&switch_code);
switch (switch_code) {
        case 1:
                if (!((pv[0] == *p2)&&(pv[1] == *(p2+1))&&(pv[2] == *(p2+2)))) {
```

```
                for (j=0;j<3;j++)
                        old_ver[num_old_ver][j] = pv[j];
                num_old_ver++;
                }
        break;
case 2:
        if (!((pv[0] == *p1)&&(pv[1] == *(p1+1))&&(pv[2] == *(p1+2)))) {
                for (j=0;j<3;j++)
                        old_ver[num_old_ver][j] = pv[j];
                num_old_ver++;
                }
}

/*
clip against the right plane of the view volumn
*/
if ( num_old_ver > 1) {
num_new_ver=0;
p1 = &(old_ver[0][0]);
p2 = &(old_ver[1][0]);
clip_right(p1,p2,pv,&switch_code);
switch (switch_code) {
        case 1:
                for (j=0;j<3;j++)
                        new_ver[num_new_ver][j] = pv[j];
                num_new_ver++;
                if (!((pv[0] == *p2)&&(pv[1] == *(p2+1))&&(pv[2] == *(p2+2)))) {
                        for (j=0;j<3;j++)
                                new_ver[num_new_ver][j] = *(p2 + j);
                        num_new_ver++;
                        }
                break;
        case 2:
                for (j=0;j<3;j++)
                        new_ver[num_new_ver][j] = *(p1 + j);
                num_new_ver++;
                if (!((pv[0] == *p1)&&(pv[1] == *(p1+1))&&(pv[2] == *(p1+2)))) {
                        for (j=0;j<3;j++)
                                new_ver[num_new_ver][j] = pv[j];
                        num_new_ver++;
                        }
                break;
        case 3:
                for (j=0;j<3;j++)
                        new_ver[num_new_ver][j] = *(p1 + j);
                num_new_ver++;
```

```
                for (j=0;j<3;j++)
                        new_ver[num_new_ver][j] = *(p2 + j);
                num_new_ver++;
        }
for (i=1;i<num_old_ver-1;i++) {
        p1 = &(old_ver[i][0]);
        p2 = &(old_ver[i+1][0]);
        clip_right(p1,p2,pv,&switch_code);
        switch (switch_code) {
                case 1:
                        for (j=0;j<3;j++)
                                new_ver[num_new_ver][j] = pv[j];
                        num_new_ver++;
                        if (!((pv[0] == *p2) && (pv[1] == *(p2+1)) && (pv[2] ==
                                        *(p2+2)))) {
                                for (j=0;j<3;j++)
                                        new_ver[num_new_ver][j] = *(p2 + j);
                                num_new_ver++;
                                }
                        break;
                case 2:
                        if (!((pv[0] == *p1) && (pv[1] == *(p1+1)) && (pv[2] ==
                                        *(p1+2)))) {
                                for (j=0;j<3;j++)
                                        new_ver[num_new_ver][j] = pv[j];
                                num_new_ver++;
                                }
                        break;
                case 3:
                        for (j=0;j<3;j++)
                                new_ver[num_new_ver][j] = *(p2 + j);
                        num_new_ver++;
                }
        }
p1 = &(old_ver[num_old_ver-1][0]);
p2 = &(old_ver[0][0]);
clip_right(p1,p2,pv,&switch_code);
switch (switch_code) {
        case 1:
                if (!((pv[0] == *p2)&&(pv[1] == *(p2+1))&&(pv[2] == *(p2+2)))) {
                        for (j=0;j<3;j++)
                                new_ver[num_new_ver][j] = pv[j];
                        num_new_ver++;
                        }
                break;
        case 2:
```

```
            if (!((pv[0] == *p1)&&(pv[1] == *(p1+1))&&(pv[2] == *(p1+2)))) {
                for (j=0;j<3;j++)
                    new_ver[num_new_ver][j] = pv[j];
                num_new_ver++;
                }
        }

/*
clip against the bottom plane of the view volumn
*/
if (num_new_ver > 1) {
num_old_ver = 0;
p1 = &(new_ver[0][0]);
p2 = &(new_ver[1][0]);
clip_bottom(p1,p2,pv,&switch_code);
switch (switch_code) {
        case 1:
            for (j=0;j<3;j++)
                old_ver[num_old_ver][j] = pv[j];
            num_old_ver++;
            if (!((pv[0] == *p2)&&(pv[1] == *(p2+1))&&(pv[2] == *(p2+2)))) {
                for (j=0;j<3;j++)
                    old_ver[num_old_ver][j] = *(p2 + j);
                num_old_ver++;
                }
            break;
        case 2:
            for (j=0;j<3;j++)
                old_ver[num_old_ver][j] = *(p1 + j);
            num_old_ver++;
            if (!((pv[0] == *p1)&&(pv[1] == *(p1+1))&&(pv[2] == *(p1+2)))) {
                for (j=0;j<3;j++)
                    old_ver[num_old_ver][j] = pv[j];
                num_old_ver++;
                }
            break;
        case 3:
            for (j=0;j<3;j++)
                old_ver[num_old_ver][j] = *(p1 + j);
            num_old_ver++;
            for (j=0;j<3;j++)
                old_ver[num_old_ver][j] = *(p2 + j);
            num_old_ver++;
        }
```

```
for (i=1;i<num_new_ver-1;i++) {
        p1 = &(new_ver[i][0]);
        p2 = &(new_ver[i+1][0]);
        clip_bottom(p1,p2,pv,&switch_code);
        switch (switch_code) {
                case 1:
                        for (j=0;j<3;j++)
                                old_ver[num_old_ver][j] = pv[j];
                        num_old_ver++;
                        if (!((pv[0] == *p2) && (pv[1] == *(p2+1)) && (pv[2] ==
                                        *(p2+2)))) {
                                for (j=0;j<3;j++)
                                        old_ver[num_old_ver][j] = *(p2 + j);
                                num_old_ver++;
                                }
                        break;
                case 2:
                        if (!((pv[0] == *p1) && (pv[1] == *(p1+1)) && (pv[2] ==
                                        *(p1+2)))) {
                                for (j=0;j<3;j++)
                                        old_ver[num_old_ver][j] = pv[j];
                                num_old_ver++;
                                }
                        break;
                case 3:
                        for (j=0;j<3;j++)
                                old_ver[num_old_ver][j] = *(p2 + j);
                        num_old_ver++;
                }
        }
p1 = &(new_ver[num_new_ver-1][0]);
p2 = &(new_ver[0][0]);
clip_bottom(p1,p2,pv,&switch_code);
switch (switch_code) {
        case 1:
                if (!((pv[0] == *p2)&&(pv[1] == *(p2+1))&&(pv[2] == *(p2+2)))) {
                        for (j=0;j<3;j++)
                                old_ver[num_old_ver][j] = pv[j];
                        num_old_ver++;
                        }
                break;
        case 2:
                if (!((pv[0] == *p1)&&(pv[1] == *(p1+1))&&(pv[2] == *(p1+2)))) {
                        for (j=0;j<3;j++)
                                old_ver[num_old_ver][j] = pv[j];
                        num_old_ver++;
```

```
                }
        }

/*
clip against the left plane of the view volumn
*/
if ( num_old_ver > 1) {
num_new_ver=0;
p1 = &(old_ver[0][0]);
p2 = &(old_ver[1][0]);
clip_left(p1,p2,pv,&switch_code);
switch (switch_code) {
        case 1:
                for (j=0;j<3;j++)
                        new_ver[num_new_ver][j] = pv[j];
                num_new_ver++;
                if (!((pv[0] == *p2)&&(pv[1] == *(p2+1))&&(pv[2] == *(p2+2)))) {
                        for (j=0;j<3;j++)
                                new_ver[num_new_ver][j] = *(p2 + j);
                        num_new_ver++;
                        }
                break;
        case 2:
                for (j=0;j<3;j++)
                        new_ver[num_new_ver][j] = *(p1 + j);
                num_new_ver++;
                if (!((pv[0] == *p1)&&(pv[1] == *(p1+1))&&(pv[2] == *(p1+2)))) {
                        for (j=0;j<3;j++)
                                new_ver[num_new_ver][j] = pv[j];
                        num_new_ver++;
                        }
                break;
        case 3:
                for (j=0;j<3;j++)
                        new_ver[num_new_ver][j] = *(p1 + j);
                num_new_ver++;
                for (j=0;j<3;j++)
                        new_ver[num_new_ver][j] = *(p2 + j);
                num_new_ver++;
        }
for (i=1;i<num_old_ver-1;i++) {
        p1 = &(old_ver[i][0]);
        p2 = &(old_ver[i+1][0]);
        clip_left(p1,p2,pv,&switch_code);
        switch (switch_code) {
```

```
                case 1:
                        for (j=0;j<3;j++)
                                new_ver[num_new_ver][j] = pv[j];
                        num_new_ver++;
                        if (!((pv[0] == *p2) && (pv[1] == *(p2+1)) && (pv[2] ==
                                *(p2+2)))) {
                                for (j=0;j<3;j++)
                                        new_ver[num_new_ver][j] = *(p2 + j);
                                num_new_ver++;
                                }
                        break;
                case 2:
                        if (!((pv[0] == *p1) && (pv[1] == *(p1+1)) && (pv[2] ==
                                *(p1+2)))) {
                                for (j=0;j<3;j++)
                                        new_ver[num_new_ver][j] = pv[j];
                                num_new_ver++;
                                }
                        break;
                case 3:
                        for (j=0;j<3;j++)
                                new_ver[num_new_ver][j] = *(p2 + j);
                        num_new_ver++;
                }
        }
p1 = &(old_ver[num_old_ver-1][0]);
p2 = &(old_ver[0][0]);
clip_left(p1,p2,pv,&switch_code);
switch (switch_code) {
        case 1:
                if (!((pv[0] == *p2)&&(pv[1] == *(p2+1))&&(pv[2] == *(p2+2)))) {
                        for (j=0;j<3;j++)
                                new_ver[num_new_ver][j] = pv[j];
                        num_new_ver++;
                        }
                break;
        case 2:
                if (!((pv[0] == *p1)&&(pv[1] == *(p1+1))&&(pv[2] == *(p1+2)))) {
                        for (j=0;j<3;j++)
                                new_ver[num_new_ver][j] = pv[j];
                        num_new_ver++;
                        }
        }
```

```
/*
clip against the front plane of the view volumn
*/
if (num_new_ver > 1) {
p1 = &(new_ver[0][0]);
p2 = &(new_ver[1][0]);
clip_front(p1,p2,pv,&switch_code);
switch (switch_code) {
        case 1:
                for (j=0;j<3;j++)
                        *(out_ver+3*(*num_out_ver)+j) = pv[j];
                (*num_out_ver)++;
                if (!((pv[0] == *p2)&&(pv[1] == *(p2+1))&&(pv[2] == *(p2+2)))) {
                        for (j=0;j<3;j++)
                                *(out_ver+3*(*num_out_ver)+j) = *(p2 + j);
                        (*num_out_ver)++;
                        }
                break;
        case 2:
                for (j=0;j<3;j++)
                        *(out_ver+3*(*num_out_ver)+j) = *(p1 + j);
                (*num_out_ver)++;
                if (!((pv[0] == *p1)&&(pv[1] == *(p1+1))&&(pv[2] == *(p1+2)))) {
                        for (j=0;j<3;j++)
                                *(out_ver+3*(*num_out_ver)+j) = pv[j];
                        (*num_out_ver)++;
                        }
                break;
        case 3:
                for (j=0;j<3;j++)
                        *(out_ver+3*(*num_out_ver)+j) = *(p1 + j);
                (*num_out_ver)++;
                for (j=0;j<3;j++)
                        *(out_ver+3*(*num_out_ver)+j) = *(p2 + j);
                (*num_out_ver)++;
        }
for (i=1;i<num_new_ver-1;i++) {
        p1 = &(new_ver[i][0]);
        p2 = &(new_ver[i+1][0]);
        clip_front(p1,p2,pv,&switch_code);
        switch (switch_code) {
                case 1:
                        for (j=0;j<3;j++)
                                *(out_ver+3*(*num_out_ver)+j) = pv[j];
                        (*num_out_ver)++;
```

```
                        if (!((pv[0] == *p2) && (pv[1] == *(p2+1)) && (pv[2] ==
                                *(p2+2)))) {
                                for (j=0;j<3;j++)
                                        *(out_ver+3*(*num_out_ver)+j) = *(p2 + j);
                                (*num_out_ver)++;
                                }
                        break;
                case 2:
                        if (!((pv[0] == *p1) && (pv[1] == *(p1+1)) && (pv[2] ==
                                        *(p1+2)))) {
                                for (j=0;j<3;j++)
                                        *(out_ver+3*(*num_out_ver)+j) = pv[j];
                                (*num_out_ver)++;
                                }
                        break;
                case 3:
                        for (j=0;j<3;j++)
                                *(out_ver+3*(*num_out_ver)+j) = *(p2 + j);
                        (*num_out_ver)++;
                }
        }
p1 = &(new_ver[num_new_ver-1][0]);
p2 = &(new_ver[0][0]);
clip_front(p1,p2,pv,&switch_code);
switch (switch_code) {
        case 1:
                if (!((pv[0] == *p2)&&&(pv[1] == *(p2+1))&&&(pv[2] == *(p2+2)))) {
                        for (j=0;j<3;j++)
                                *(out_ver+3*(*num_out_ver)+j) = pv[j];
                        (*num_out_ver)++;
                        }
                break;
        case 2:
                if (!((pv[0] == *p1)&&&(pv[1] == *(p1+1))&&&(pv[2] == *(p1+2)))) {
                        for (j=0;j<3;j++)
                                *(out_ver+3*(*num_out_ver)+j) = pv[j];
                        (*num_out_ver)++;
                        }
        }       }       }       }       }       }
}
```

```
/*
clip a line aginst the top plane of the canonical view volumn
*/
clip_top(p1,p2,pv,out_code)
float           *p1,*p2,*pv;
int             *out_code;
{
int             p1_test,p2_test;
p1_test = (*(p1+1) > *(p1+2));
p2_test = (*(p2+1) > *(p2+2));
if ( p1_test && p2_test )
        *out_code = 0;
else {
        if ( p1_test ) {
                interset(p1,p2,pv,1);
                *out_code = 1;
                }
        else {
                if ( p2_test ) {
                        interset(p1,p2,pv,1);
                        *out_code = 2;
                        }
                else
                        *out_code = 3;
                }
        }
}

/*
clip a line aginst the right plane of the canonical view volumn
*/
clip_right(p1,p2,pv,out_code)
float           *p1,*p2,*pv;
int             *out_code;
{
int             p1_test,p2_test;
p1_test = (*p1 > *(p1+2));
p2_test = (*p2 > *(p2+2));
if ( p1_test && p2_test )
        *out_code = 0;
else {
        if ( p1_test ) {
                interset(p1,p2,pv,2);
                *out_code = 1;
                }
        else {
```

```
                if ( p2_test ) {
                        interset(p1,p2,pv,2);
                        *out_code = 2;
                        }
                else
                        *out_code = 3;
                }
        }
}

/*
clip a line aginst the bottom plane of the canonical view volumn
*/
clip_bottom(p1,p2,pv,out_code)
float           *p1,*p2,*pv;
int             *out_code;
{
int             p1_test,p2_test;
p1_test = (*(p1+1) < -*(p1+2));
p2_test = (*(p2+1) < -*(p2+2));
if ( p1_test && p2_test )
        *out_code = 0;
else {
        if ( p1_test ) {
                interset(p1,p2,pv,3);
                *out_code = 1;
                }
        else {
                if ( p2_test ) {
                        interset(p1,p2,pv,3);
                        *out_code = 2;
                        }
                else
                        *out_code = 3;
                }
        }
}

/*
clip a line aginst the left plane of the canonical view volumn
*/
clip_left(p1,p2,pv,out_code)
float           *p1,*p2,*pv;
int             *out_code;
{
int             p1_test,p2_test;
```

```
p1_test = (*p1 < -*(p1+2));
p2_test = (*p2 < -*(p2+2));
if ( p1_test && p2_test )
        *out_code = 0;
else {
        if ( p1_test ) {
                interset(p1,p2,pv,4);
                *out_code = 1;
                }
        else {
                if ( p2_test ) {
                        interset(p1,p2,pv,4);
                        *out_code = 2;
                        }
                else
                        *out_code = 3;
                }
        }
}

/*
clip a line aginst the front plane of the canonical view volumn
*/
clip_front(p1,p2,pv,out_code)
float           *p1,*p2,*pv;
int             *out_code;
{
int             p1_test,p2_test;
p1_test = (*(p1+2) < 1.0);
p2_test = (*(p2+2) < 1.0);
if ( p1_test && p2_test )
        *out_code = 0;
else {
        if ( p1_test ) {
                interset(p1,p2,pv,5);
                *out_code = 1;
                }
        else {
                if ( p2_test ) {
                        interset(p1,p2,pv,5);
                        *out_code = 2;
                        }
                else
                        *out_code = 3;
                }
        }
```

```
}

/*
subroutine computes the intersection point of a line and a specified
view plane.
*/
interset(p1,p2,pv,plane_code)
float           *p1,*p2,*pv;
int             plane_code;
{
float           t;
switch (plane_code) {
        case 1:
                t = (*(p1+2)-*(p1+1)) / ((*(p2+1)-*(p1+1)) - (*(p2+2)-*(p1+2)));
                *pv = (*p2 - *p1) * t + *p1;
                *(pv+1) = *(pv+2) = (*(p2+1)-*(p1+1)) * t + *(p1+1);
                break;
        case 2:
                t = (*(p1+2)-*p1) / ((*p2-*p1) - (*(p2+2)-*(p1+2)));
                *pv = *(pv+2) = (*p2 - *p1) * t + *p1;
                *(pv+1) = (*(p2+1)-*(p1+1)) * t + *(p1+1);
                break;
        case 3:
                t = (*(p1+2)+*(p1+1)) / ((*(p1+1)-*(p2+1)) - (*(p2+2)-*(p1+2)));
                *pv = (*p2 - *p1) * t + *p1;
                *(pv+1) = (*(p2+1)-*(p1+1)) * t + *(p1+1);
                *(pv+2) = -*(pv+1);
                break;
        case 4:
                t = (*(p1+2)+*p1) / ((*p1-*p2) - (*(p2+2)-*(p1+2)));
                *pv = (*p2 - *p1) * t + *p1;
                *(pv+1) = (*(p2+1)-*(p1+1)) * t + *(p1+1);
                *(pv+2) = -*pv;
                break;
        case 5:
                t = (1.0 - *(p1+2)) / (*(p2+2)-*(p1+2));
                *pv = (*p2 - *p1) * t + *p1;
                *(pv+1) = (*(p2+1)-*(p1+1)) * t + *(p1+1);
                *(pv+2) = 1.0;
        }
}
```

```
/*
set_3d_rot_matrix --
        sets up a matrix for rotation about an arbitrary axis in the homogeneous
        coordinate system
        point : a point at which the axis passing through
        vector : direction of the axis
        theta : rotation angle
*/
#include <math.h>
#include "variable.h"

set_3d_rot_matrix(point,vector,theta)
float           point[],vector[],theta;
{
extern float    rot_3d[4][4];
float           a_square,b_square,c_square,v_square;
float           ctheta,stheta,one_ctheta,k1,k2,k3;

theta = theta * 3.14159 / 180;
norm_vec_3(vector);
a_square = vector[0]*vector[0];
b_square = vector[1]*vector[1];
c_square = vector[2]*vector[2];
v_square = b_square + c_square;
ctheta = cos(theta);
stheta = sin(theta);
one_ctheta = 1 - ctheta;
if (v_square == 0) {
        k1 = k3 = 1;
        k2 = 0;
        }
else {
        k1 = (a_square * b_square + c_square) / v_square;
        k2 = (vector[1]*vector[2]*(a_square-1)) / v_square;
        k3 = (a_square*c_square + b_square) / v_square;
        }
rot_3d[0][0] = v_square * ctheta + a_square;
rot_3d[0][1] = vector[0] * vector[1] * one_ctheta - vector[2] * stheta;
rot_3d[0][2] = vector[0] * vector[2] * one_ctheta + vector[1] * stheta;
rot_3d[1][0] = vector[0] * vector[1] * one_ctheta + vector[2] * stheta;
rot_3d[1][1] = k1 * ctheta + b_square;
rot_3d[1][2] = k2 * ctheta - vector[0] * stheta + vector[1] * vector[2];
rot_3d[2][0] = vector[0] * vector[2] * one_ctheta - vector[1] * stheta;
rot_3d[2][1] = k2 * ctheta + vector[0] * stheta + vector[1] * vector[2];
rot_3d[2][2] = k3 * ctheta + c_square;
rot_3d[0][3] = rot_3d[1][3] = rot_3d[2][3] = 0;
```

```
rot_3d[3][0] = point[0] - point[0] * rot_3d[0][0] - point[1] * rot_3d[1][0] -
               point[2] * rot_3d[2][0];
rot_3d[3][1] = point[1] - point[0] * rot_3d[0][1] - point[1] * rot_3d[1][1] -
               point[2] * rot_3d[2][1];
rot_3d[3][2] = point[2] - point[0] * rot_3d[0][2] - point[1] * rot_3d[1][2] -
               point[2] * rot_3d[2][2];
rot_3d[3][3] = 1;
}
```

```
/*
drawline --
        this subroutine draws a straight line from pt1 to pt2
*/
#include <bgraph.h>
#include "variable.h"

drawline(pt1,pt2)
float *pt1,*pt2;
{

PT              a,b;
float           temp;
extern int      I_CENTER,J_CENTER,I_PIXEL,J_PIXEL;
int             logic_flag;

temp = *pt1 * I_PIXEL;
logic_flag = (sign(temp)) + 1;
if (logic_flag)
        a.x = round_pos(temp);
else
        a.x = round_neg(temp);
temp = *(pt1+1) * J_PIXEL;
logic_flag = (sign(temp)) + 1;
if (logic_flag)
        a.y = round_pos(temp);
else
        a.y = round_neg(temp);
temp = *pt2 * I_PIXEL;
logic_flag = (sign(temp)) + 1;
if (logic_flag)
        b.x = round_pos(temp);
else
        b.x = round_neg(temp);
temp = *(pt2+1) * J_PIXEL;
logic_flag = (sign(temp)) + 1;
if (logic_flag)
        b.y = round_pos(temp);
else
        b.y = round_neg(temp);
a.x=a.x+I_CENTER;
a.y= -a.y+J_CENTER;
b.x=b.x+I_CENTER;
b.y= -b.y+J_CENTER;
grline(&a,&b,15);
}
```

```
/*
rand_nor.c --
        The univariate normal distribution random number generator - Butcher-Kahn's
        first method.
*/

#include        <stdlib.h>
#include        <math.h>
#define         sign(x)         (x < 0) ? -1 : 1

double rand_normal(alpha)
double          alpha;
{
double          zeta1,zeta2,zeta3,y,z,g;
do {
        zeta1 = (double)rand()/32767.;
        zeta2 = (double)rand()/32767.;
        if (zeta1 < 0.001) zeta1 = 0.001;
        y = - log(zeta1);
        z = (y - 1.)*(y - 1.) / 2.;
        g = exp(-z);
        } while (zeta2 > g);
zeta3 = (double)rand()/32767.;
z = sign(zeta3 - 0.5);
y *= z;
return(y*alpha);
}
```

```
/*
norm_per.c --
        sets up the normalization matrix nper
*/

#include <math.h>

set_up_nper(mode)
int     mode;
{
extern float            cop[3];
extern float            view_plane_i[3],view_plane_j[3],view_plane_k[3];
extern float            nper[4][4];
extern float            h_view_angle,v_view_angle;
extern float            h_scan_angle,v_scan_angle;
float                   left_mat[4][4],right_mat[4][4],res_mat[4][4];
float                   theta1,theta2;
int                     i;

/*
set up translation matrix
*/
unit_matrix(left_mat,4);
for (i=0;i<3;i++)
        left_mat[3][i] = -cop[i];

/*
set up rotational matrix so that the i and j axes of the view coordinates
are aligned with the x and y axes of the world coordinates
*/
unit_matrix(right_mat,4);
for (i=0;i<3;i++)
        right_mat[i][0] = view_plane_i[i];
for (i=0;i<3;i++)
        right_mat[i][1] = view_plane_j[i];
for (i=0;i<3;i++)
        right_mat[i][2] = -view_plane_k[i];

/*
multiply the translation and rotation matrices
*/
mdotm(left_mat,4,4,right_mat,4,4,res_mat);
```

```
/*
convert the right-handed world coordinates to a left-handed coordinates
*/
unit_matrix(right_mat,4);
right_mat[2][2] = -1.0;
mdotm(res_mat,4,4,right_mat,4,4,left_mat);

/*
scale so that the view volumn becomes a canonical view volumn
*/
unit_matrix(right_mat,4);
if (mode == 1) {
        right_mat[0][0] = 1 / tan(h_view_angle*3.14159/360);
        right_mat[1][1] = 1 / tan(v_view_angle*3.14159/360);
        }
else {
        theta1 = v_scan_angle * 3.14159 / 360.;
        theta2 = h_scan_angle * 3.14159 / 360.;
        right_mat[0][0] = cos(theta1) / tan(theta2);
        right_mat[1][1] = 1 / tan(theta1);
        }
mdotm(left_mat,4,4,right_mat,4,4,nper);
}
```

```
/*
shad.c --
        This subroutine generates range and reflectance data.

        by Chris K. WU, May 1988
*/
#include      "variable.h"
#include      <math.h>
#define       RUN_SHAD
#define       ALPHA       0.25
#define       ITA         0.10
#define       LO          0.30

shad_depth(refresh_buf,depth_buf)
float   huge          *refresh_buf;
float huge            *depth_buf;
{
extern int            las_i_pixel,las_j_pixel,add_noise;
extern int            h_pixel,v_pixel;
extern int            num_clip_ver;
extern float          clip_ver[MAXPVERT*2][3];
extern float          v_scan_angle,h_scan_angle;
extern float          lgt[3];
extern float          lambda,ar,ft,t,lambdam;
struct et_slot        et_entry[MAXPVERT*2],*et[MAX_Y_SCAN],*aet;
struct et_slot        *address1,*address2,**j_start_address;
struct h_edge         *het[MAX_Y_SCAN],het_entry[MAXPVERT/2];
struct h_edge         *h_address1,*h_address2;
struct view_point     xy_ver[MAXPVERT*2];
int                   i,j,itemp,aet_count,jindex,y_diff,delete_aet;
int                   et_index,het_index,flat_edge;
int                   et_array[MAX_Y_SCAN],het_array[MAX_Y_SCAN],logic_flag;
int                   v1,v2,v3,vtmp;
float                 planeq[4];
float                 x,y,z,length;
float                 h_angle;
float                 ftemp,*left_vec,*right_vec;
float                 right_mat[3][3],poly_3d[MAXPVERT*2][3];
float                 theta1,theta2,delta_x,delta_y;
float                 delta_theta,delta_phi;
float                 phi,x_prime,y_prime,d_length,l_angle;
float                 x_value,y_value,z_value;
float                 z_axis[3];
float                 snr,sigma,sigma_r;
double                rand_normal();
```

```
#ifdef  RUN_SHAD
float   huge            *z_index;
float huge              *r_index;
#endif

aet_count = 0;
het_index = 0;

/*
initialize edge table
*/
for (i=0; i<v_pixel+1 ; i++) {
        et_array[i] = 0;
        het_array[i] = 0;
        }

/*
scale clipped polygon back to the original dimension
*/
unit_matrix(right_mat,3);
h_angle = h_scan_angle * 3.14159 / 360.;
theta1 = v_scan_angle * 3.14159 / 360.;
theta2 = h_scan_angle * 3.14159 / 360.;
right_mat[0][0] = tan(theta2) / cos(theta1);
right_mat[1][1] = tan(theta1);
for (i=0;i<num_clip_ver;i++) {
        left_vec = clip_ver[i];
        right_vec = &(poly_3d[i][0]);
        vdotm(left_vec,right_mat,3,3,right_vec);
        }

/*
compute the plane equation of this polygon using the last
three vertices
plane eq. :             Ax+By+Cz+D=0
*/
        planeq[0] =planeq[1] =planeq[2] =planeq[3] =0.0;
        v1 = num_clip_ver-1;
        v2 = num_clip_ver-2;
        v3 = num_clip_ver-3;
        for(j=0;j<3;j++) {
                planeq[0] += poly_3d[v1][1] *
                        (poly_3d[v2][2] - poly_3d[v3][2]);
                planeq[1] += poly_3d[v1][0] *
                        (poly_3d[v3][2] - poly_3d[v2][2]);
                planeq[2] += poly_3d[v1][0] *
```

```
                (poly_3d[v2][1] - poly_3d[v3][1]);
            planeq[3] += poly_3d[v1][0] *
                ((poly_3d[v3][1] * poly_3d[v2][2]) -
                 (poly_3d[v2][1] * poly_3d[v3][2]));
            vtmp = v1;
            v1 = v2;
            v2 = v3;
            v3 = vtmp;
            }
        x = planeq[0];
        y = planeq[1];
        z = planeq[2];
        length = sqrt(x * x + y * y + z * z);
        planeq[0] /= length;
        planeq[1] /= length;
        planeq[2] /= length;
        planeq[3] /= length;
if (    (fabs(planeq[0]) <= EPSILON) && ((fabs(planeq[1]) - 1.) <= EPSILON)
        && (fabs(planeq[2]) <= EPSILON) && (fabs(planeq[3]) <= EPSILON) )
        goto skip_shad;

/*
assign reflectance value to the cosine of the angle
between the plane normal vector and the z-axis
*/
z_axis[0] = z_axis[1] = 0;
z_axis[2] = 1;
l_angle = vdotv(planeq,z_axis,3);
l_angle = (float) fabs(l_angle);

/*
compute the signal-noise-ratio
*/
if (add_noise) {
        if ( l_angle > 0.01) {
                snr = ALPHA*ITA*lambda*1e-9*ar*ft*1e-3*t*LO*l_angle;
                snr /= (3.14159*1.9878e-25*h_pixel*v_pixel);
                snr = sqrt(snr);
                sigma_r = 3.e2/(8.8858*snr*lambdam);
                }
        else sigma_r = 0.2;
        }

delta_y = right_mat[1][1] / las_j_pixel;
delta_theta = (h_scan_angle*3.14159) / ((float)h_pixel*180);
delta_phi = (v_scan_angle*3.14159) / ((float)v_pixel*180);
```

```
/*
calculate the projection of the clipped polygon on the view plane
*/
for (i=0; i<num_clip_ver; i++) {
        xy_ver[i].x = poly_3d[i][0] / poly_3d[i][2];
        ftemp = (poly_3d[i][1]/poly_3d[i][2]) * (las_j_pixel/right_mat[1][1]);
        logic_flag = (sign(ftemp))+1;
        if (logic_flag) {
                itemp = round_pos(ftemp);
                xy_ver[i].y = -itemp + las_j_pixel;
                }
        else    {
                itemp = round_neg(ftemp);
                xy_ver[i].y = -itemp + las_j_pixel;
                }
        }

/*
set up the edge table
*/
if ( xy_ver[0].y < xy_ver[1].y ) {
        et_entry[0].y_max = xy_ver[1].y;
        y_diff = xy_ver[1].y - xy_ver[0].y;
        et_entry[0].m = (xy_ver[1].x-xy_ver[0].x) / y_diff;
        if (xy_ver[0].y < xy_ver[num_clip_ver-1].y) {
                et_entry[0].x_min = xy_ver[0].x;
                itemp = xy_ver[0].y;
                }
        else {
                et_entry[0].x_min = xy_ver[0].x+et_entry[0].m;
                itemp = xy_ver[0].y+1;
                }
        et[itemp] = &(et_entry[0]);
        et_array[itemp]++;
        flat_edge = 0;
        et_index = 1;
        }
else {
        if (xy_ver[0].y == xy_ver[1].y) {
                itemp = xy_ver[0].y;
                if (
                        (
                (xy_ver[num_clip_ver-1].y>xy_ver[0].y)&&(xy_ver[2].y>xy_ver[1].y) )
                        ||
                        (
                (xy_ver[num_clip_ver-1].y<xy_ver[0].y)&&(xy_ver[2].y<xy_ver[1].y) )
```

```
                ) {
                et_entry[0].y_max = et_entry[1].y_max = xy_ver[0].y;
                et_entry[0].x_min = xy_ver[0].x;
                et_entry[1].x_min = xy_ver[1].x;
                et_entry[0].m = et_entry[1].m = 0;
                et_entry[0].next_slot = &(et_entry[1]);
                et_array[itemp] = 2;
                et_index = 2;
                }
        else {
                et_entry[0].y_max = xy_ver[0].y;
                et_entry[0].x_min = xy_ver[0].x;
                et_entry[0].m = 0;
                et_array[itemp] = 1;
                et_index = 1;
                }
        if(xy_ver[0].x > xy_ver[1].x) {
                het_entry[0].x_min = xy_ver[1].x;
                het_entry[0].x_max = xy_ver[0].x;
                }
        else {
                het_entry[0].x_min = xy_ver[0].x;
                het_entry[0].x_max = xy_ver[1].x;
                }
        het_index++;
        het_array[itemp] = 1;
        het[itemp] = &(het_entry[0]);
        et[itemp] = &(et_entry[0]);
        flat_edge = 1;
        }
else {
        et_entry[0].y_max = xy_ver[0].y;
        y_diff = xy_ver[0].y - xy_ver[1].y;
        et_entry[0].m=(xy_ver[0].x-xy_ver[1].x)/y_diff;
        if (xy_ver[1].y < xy_ver[2].y) {
                et_entry[0].x_min = xy_ver[1].x;
                itemp = xy_ver[1].y;
                }
        else {
                et_entry[0].x_min = xy_ver[1].x + et_entry[0].m;
                itemp = xy_ver[1].y + 1;
                }
        et[itemp] = &(et_entry[0]);
        et_array[itemp]++;
        et_index = 1;
        flat_edge = 0;
```

```
            }
        }

for (i=1; i<num_clip_ver-2; i++) {
        if ( xy_ver[i].y < xy_ver[i+1].y ) {
                et_entry[et_index].y_max = xy_ver[i+1].y;
                y_diff = xy_ver[i+1].y - xy_ver[i].y;
                et_entry[et_index].m = (xy_ver[i+1].x-xy_ver[i].x) / y_diff;
                if (xy_ver[i].y < xy_ver[i-1].y) {
                        et_entry[et_index].x_min = xy_ver[i].x;
                        itemp = xy_ver[i].y;
                        }
                else {
                        et_entry[et_index].x_min = xy_ver[i].x+et_entry[et_index].m;
                        itemp = xy_ver[i].y+1;
                        }
                if (!et_array[itemp])
                        et[itemp] = &(et_entry[et_index]);
                else {
                        address1 = et[itemp];
                        for (j = 0; j<et_array[itemp]-1 ; j++) {
                                address2 = address1->next_slot;
                                address1 = address2;
                                }
                        address1->next_slot = &(et_entry[et_index]);
                        }
                et_array[itemp]++;
                flat_edge = 0;
                et_index++;
                }
        else {
                if (xy_ver[i].y == xy_ver[i+1].y ) {
                        if (xy_ver[i-1].y < xy_ver[i].y)      /*shorten previous edge*/
                                et_entry[et_index-1].y_max--;
                        itemp = xy_ver[i].y;
                        if (!et_array[itemp])
                                et[itemp] = &(et_entry[et_index]);
                        else {
                                address1 = et[itemp];
                                for (j = 0; j<et_array[itemp]-1 ; j++) {
                                        address2 = address1->next_slot;
                                        address1 = address2;
                                        }
                                address1->next_slot = &(et_entry[et_index]);
                                }
                        if (!het_array[itemp])
```

```
                het[itemp] = &(het_entry[het_index]);
        else {
                h_address1 = het[itemp];
                for (j = 0; j<het_array[itemp]-1 ; j++) {
                        h_address2 = h_address1->next_edge;
                        h_address1 = h_address2;
                        }
                h_address1->next_edge = &(het_entry[het_index]);
                }
        if (
                (
        (xy_ver[i-1].y>xy_ver[i].y)&&(xy_ver[i+2].y>xy_ver[i+1].y) )
                ||
                (
        (xy_ver[i-1].y<xy_ver[i].y)&&(xy_ver[i+2].y<xy_ver[i+1].y) )
                ) {
et_entry[et_index].y_max=et_entry[et_index+1].y_max=xy_ver[i].y;
                et_entry[et_index].x_min = xy_ver[i].x;
                et_entry[et_index+1].x_min = xy_ver[i+1].x;
                et_entry[et_index].m = et_entry[et_index+1].m = 0;
                et_entry[et_index].next_slot = &(et_entry[et_index+1]);
                et_array[itemp] = et_array[itemp] + 2;
                et_index = et_index + 2;
                }
        else {
                et_entry[et_index].y_max = xy_ver[i].y;
                et_entry[et_index].x_min = xy_ver[i].x;
                et_entry[et_index].m = 0;
                et_array[itemp]++;
                et_index++;
                }
        if(xy_ver[i].x > xy_ver[i+1].x) {
                het_entry[het_index].x_min = xy_ver[i+1].x;
                het_entry[het_index].x_max = xy_ver[i].x;
                }
        else {
                het_entry[het_index].x_min = xy_ver[i].x;
                het_entry[het_index].x_max = xy_ver[i+1].x;
                }
        het_index++;
        het_array[itemp]++;
        flat_edge = 1;
        }
else {
        if (flat_edge)
                et_entry[et_index].y_max = xy_ver[i].y - 1;
```

```
                    else
                        et_entry[et_index].y_max = xy_ver[i].y;
                    y_diff = xy_ver[i].y - xy_ver[i+1].y;
                    et_entry[et_index].m = (xy_ver[i].x-xy_ver[i+1].x) / y_diff;
                    if ( xy_ver[i+1].y < xy_ver[i+2].y ) {
                        et_entry[et_index].x_min = xy_ver[i+1].x;
                        itemp = xy_ver[i+1].y;
                        }
                    else {
                et_entry[et_index].x_min = xy_ver[i+1].x + et_entry[et_index].m;
                        itemp = xy_ver[i+1].y + 1;
                        }
                    if (!et_array[itemp])
                        et[itemp] = &(et_entry[et_index]);
                    else {
                        address1 = et[itemp];
                        for (j = 0; j<et_array[itemp]-1 ; j++) {
                            address2 = address1->next_slot;
                            address1 = address2;
                            }
                        address1->next_slot = &(et_entry[et_index]);
                        }
                    et_array[itemp]++;
                    flat_edge = 0;
                    et_index ++;
                    }
                }
            }

if ( xy_ver[num_clip_ver-2].y < xy_ver[num_clip_ver-1].y ) {
        et_entry[et_index].y_max = xy_ver[num_clip_ver-1].y;
        y_diff = xy_ver[num_clip_ver-1].y - xy_ver[num_clip_ver-2].y;
        et_entry[et_index].m = (xy_ver[num_clip_ver-1].x
                -xy_ver[num_clip_ver-2].x) / y_diff;
        if (xy_ver[num_clip_ver-2].y < xy_ver[num_clip_ver-3].y) {
                et_entry[et_index].x_min = xy_ver[num_clip_ver-2].x;
                itemp = xy_ver[num_clip_ver-2].y;
                }
        else {
                et_entry[et_index].x_min =
                        xy_ver[num_clip_ver-2].x+et_entry[et_index].m;
                itemp = xy_ver[num_clip_ver-2].y+1;
                }
        if (!et_array[itemp])
                et[itemp] = &(et_entry[et_index]);
        else {
```

```
                address1 = et[itemp];
                for (j = 0; j<et_array[itemp]-1 ; j++) {
                        address2 = address1->next_slot;
                        address1 = address2;
                        }
                address1->next_slot = &(et_entry[et_index]);
                }
        et_array[itemp]++;
        flat_edge = 0;
        et_index++;
        }
else {
        if (xy_ver[num_clip_ver-2].y == xy_ver[num_clip_ver-1].y ) {
                if(xy_ver[num_clip_ver-3].y<xy_ver[num_clip_ver-2].y)
                        et_entry[et_index-1].y_max--;       /*shorten previous edge*/
                itemp = xy_ver[num_clip_ver-2].y;
                if (!et_array[itemp])
                                et[itemp] = &(et_entry[et_index]);
                else {
                        address1 = et[itemp];
                        for (j = 0; j<et_array[itemp]-1 ; j++) {
                                address2 = address1->next_slot;
                                address1 = address2;
                                }
                        address1->next_slot = &(et_entry[et_index]);
                        }
                if (!het_array[itemp])
                        het[itemp] = &(het_entry[het_index]);
                else {
                        h_address1 = het[itemp];
                        for (j = 0; j<het_array[itemp]-1 ; j++) {
                                h_address2 = h_address1->next_edge;
                                h_address1 = h_address2;
                                }
                        h_address1->next_edge = &(het_entry[het_index]);
                        }

                if (
                        ( (xy_ver[num_clip_ver-3].y>xy_ver[num_clip_ver-2].y)
                                &&(xy_ver[0].y>xy_ver[num_clip_ver-1].y) )
                        ||
                        ( (xy_ver[num_clip_ver-3].y<xy_ver[num_clip_ver-2].y)
                                &&(xy_ver[0].y<xy_ver[num_clip_ver-1].y) )
                        ) {
                        et_entry[et_index].y_max = et_entry[et_index+1].y_max
                                = xy_ver[num_clip_ver-2].y;
```

```
                et_entry[et_index].x_min = xy_ver[num_clip_ver-2].x;
                et_entry[et_index+1].x_min = xy_ver[num_clip_ver-1].x;
                et_entry[et_index].m = et_entry[et_index+1].m = 0;
                et_entry[et_index].next_slot = &(et_entry[et_index+1]);
                et_array[itemp] = et_array[itemp] + 2;
                et_index = et_index + 2;
                }
        else {
                et_entry[et_index].y_max = xy_ver[num_clip_ver-2].y;
                et_entry[et_index].x_min = xy_ver[num_clip_ver-2].x;
                et_entry[et_index].m = 0;
                et_array[itemp]++;
                et_index++;
                }
        if(xy_ver[num_clip_ver-2].x > xy_ver[num_clip_ver-1].x) {
                het_entry[het_index].x_min = xy_ver[num_clip_ver-1].x;
                het_entry[het_index].x_max = xy_ver[num_clip_ver-2].x;
                }
        else {
                het_entry[het_index].x_min = xy_ver[num_clip_ver-2].x;
                het_entry[het_index].x_max = xy_ver[num_clip_ver-1].x;
                }
        het_index++;
        het_array[itemp]++;
        flat_edge = 1;
        }
else {
        if (flat_edge)
                et_entry[et_index].y_max = xy_ver[num_clip_ver-2].y - 1;
        else
                et_entry[et_index].y_max = xy_ver[num_clip_ver-2].y;
        y_diff = xy_ver[num_clip_ver-2].y - xy_ver[num_clip_ver-1].y;
        et_entry[et_index].m = (xy_ver[num_clip_ver-2].x
                -xy_ver[num_clip_ver-1].x) / y_diff;
        if ( xy_ver[num_clip_ver-1].y < xy_ver[0].y ) {
                et_entry[et_index].x_min = xy_ver[num_clip_ver-1].x;
                itemp = xy_ver[num_clip_ver-1].y;
                }
        else {
                et_entry[et_index].x_min = xy_ver[num_clip_ver-1].x
                        + et_entry[et_index].m;
                itemp = xy_ver[num_clip_ver-1].y + 1;
                }
        if (!et_array[itemp])
                et[itemp] = &(et_entry[et_index]);
        else {
```

```
                    address1 = et[itemp];
                    for (j = 0; j<et_array[itemp]-1 ; j++) {
                            address2 = address1->next_slot;
                            address1 = address2;
                            }
                    address1->next_slot = &(et_entry[et_index]);
                    }
            et_array[itemp]++;
            flat_edge = 0;
            et_index ++;
            }
    }

if ( xy_ver[num_clip_ver-1].y < xy_ver[0].y ) {
    et_entry[et_index].y_max = xy_ver[0].y;
    y_diff = xy_ver[0].y - xy_ver[num_clip_ver-1].y;
    et_entry[et_index].m = (xy_ver[0].x
            -xy_ver[num_clip_ver-1].x) / y_diff;
    if (xy_ver[num_clip_ver-1].y < xy_ver[num_clip_ver-2].y) {
            et_entry[et_index].x_min = xy_ver[num_clip_ver-1].x;
            itemp = xy_ver[num_clip_ver-1].y;
            }
    else {
            et_entry[et_index].x_min =
                    xy_ver[num_clip_ver-1].x+et_entry[et_index].m;
            itemp = xy_ver[num_clip_ver-1].y+1;
            }
    if (!et_array[itemp])
            et[itemp] = &(et_entry[et_index]);
    else {
            address1 = et[itemp];
            for (j = 0; j<et_array[itemp]-1 ; j++) {
                    address2 = address1->next_slot;
                    address1 = address2;
                    }
            address1->next_slot = &(et_entry[et_index]);
            }
    et_array[itemp]++;
    flat_edge = 0;
    et_index++;
    }
else {
    if (xy_ver[num_clip_ver-1].y == xy_ver[0].y ) {
            if (xy_ver[num_clip_ver-2].y < xy_ver[num_clip_ver-1].y)
                    et_entry[et_index-1].y_max--;       /*shorten previous edge*/
            itemp = xy_ver[num_clip_ver-1].y;
```

```
if (!et_array[itemp])
        et[itemp] = &(et_entry[et_index]);
else {
        address1 = et[itemp];
        for (j = 0; j<et_array[itemp]-1 ; j++) {
                address2 = address1->next_slot;
                address1 = address2;
                }
        address1->next_slot = &(et_entry[et_index]);
        }
if (!het_array[itemp])
        het[itemp] = &(het_entry[het_index]);
else {
        h_address1 = het[itemp];
        for (j = 0; j<het_array[itemp]-1 ; j++) {
                h_address2 = h_address1->next_edge;
                h_address1 = h_address2;
                }
        h_address1->next_edge = &(het_entry[het_index]);
        }

if (
        ( (xy_ver[num_clip_ver-2].y>xy_ver[num_clip_ver-1].y)
                &&(xy_ver[1].y>xy_ver[0].y) )
        ||
        ( (xy_ver[num_clip_ver-2].y<xy_ver[num_clip_ver-1].y)
                &&(xy_ver[1].y<xy_ver[0].y) )
        ) {
        et_entry[et_index].y_max = et_entry[et_index+1].y_max
                = xy_ver[num_clip_ver-1].y;
        et_entry[et_index].x_min = xy_ver[num_clip_ver-1].x;
        et_entry[et_index+1].x_min = xy_ver[0].x;
        et_entry[et_index].m = et_entry[et_index+1].m = 0;
        et_entry[et_index].next_slot = &(et_entry[et_index+1]);
        et_array[itemp] = et_array[itemp] + 2;
        et_index = et_index + 2;
        }
else {
        et_entry[et_index].y_max = xy_ver[num_clip_ver-1].y;
        et_entry[et_index].x_min = xy_ver[num_clip_ver-1].x;
        et_entry[et_index].m = 0;
        et_array[itemp]++;
        et_index++;
        }
if(xy_ver[num_clip_ver-1].x > xy_ver[0].x) {
        het_entry[het_index].x_min = xy_ver[0].x;
```

```
                het_entry[het_index].x_max = xy_ver[num_clip_ver-1].x;
                }
        else {
                het_entry[het_index].x_min = xy_ver[num_clip_ver-1].x;
                het_entry[het_index].x_max = xy_ver[0].x;
                }
        het_index++;
        het_array[itemp]++;
        flat_edge = 1;
        }
else {
        if (flat_edge)
                et_entry[et_index].y_max = xy_ver[num_clip_ver-1].y - 1;
        else
                et_entry[et_index].y_max = xy_ver[num_clip_ver-1].y;
        y_diff = xy_ver[num_clip_ver-1].y - xy_ver[0].y;
        et_entry[et_index].m = (xy_ver[num_clip_ver-1].x
                -xy_ver[0].x) / y_diff;
        if ( xy_ver[0].y < xy_ver[1].y ) {
                et_entry[et_index].x_min = xy_ver[0].x;
                itemp = xy_ver[0].y;
                }
        else {
                et_entry[et_index].x_min = xy_ver[0].x
                        + et_entry[et_index].m;
                itemp = xy_ver[0].y + 1;
                }
        if (!et_array[itemp])
                et[itemp] = &(et_entry[et_index]);
        else {
                address1 = et[itemp];
                for (j = 0; j<et_array[itemp]-1 ; j++) {
                        address2 = address1->next_slot;
                        address1 = address2;
                        }
                address1->next_slot = &(et_entry[et_index]);
                }
        et_array[itemp]++;
        flat_edge = 0;
        et_index ++;
        }
}
```

```
/*
start the scan line converting process
*/
for (i=0;i<v_pixel+1;i++) {
/*
duplicate edge points from edge table to active edge table
*/
        if (et_array[i]) {
                if ( aet_count ) {
                        address1 = aet;
                        for (j = 0; j<aet_count-1 ; j++) {
                                address2 = address1->next_slot;
                                address1 = address2;
                                }
                        address1->next_slot = et[i];
                        }
                else
                        aet = et[i];
                aet_count += et_array[i];
                sort_aet(&aet,aet_count);
                }
/*
scan line converting using the active edge table
*/
        if (aet_count) {
                delete_aet = 0;
                address1 = aet;
                y_prime = (las_j_pixel - i) * delta_y;
                for (j=0;j<aet_count/2;j++) {
                        address2 = address1->next_slot;
                        ftemp = sqrt( address1->x_min*address1->x_min
                                                        + y_prime*y_prime + 1.);
                        theta1 = (asin(address1->x_min / ftemp));
                        ftemp = sqrt( address2->x_min*address2->x_min
                                                        + y_prime*y_prime + 1.);
                        theta2 = (asin(address2->x_min / ftemp));
                        phi = (las_j_pixel - i) * delta_phi;
                        if (theta1 < h_angle ) {
                                if (theta1 < -h_angle)
                                        theta1 = -h_angle;
                                if (theta2 > h_angle)
                                        theta2 = h_angle;
                                ftemp = theta1 / delta_theta;
                                logic_flag = (sign(ftemp))+1;
                                if (logic_flag) {
                                        itemp = round_pos(ftemp);
```

```
                        jindex = itemp + las_i_pixel;
                        }
                    else {
                        itemp = round_neg(ftemp);
                        jindex = itemp + las_i_pixel;
                        }
#ifdef RUN_SHAD
                    z_index =
                    depth_buf+(long)i*(long)(h_pixel+1)+(long)jindex;
                    r_index =
                    refresh_buf+(long)i*(long)(h_pixel+1)+(long)jindex;
#endif
                    do {
                        x_prime = tan(theta1) / cos(phi);
                        z_value = -planeq[3] / (planeq[0] * x_prime +
                            planeq[1] *  y_prime + planeq[2]);
                        x_value = x_prime * z_value;
                        y_value = y_prime * z_value;
                        d_length    =    sqrt(x_value*x_value    +
                            y_value*y_value + z_value*z_value);
                        if (add_noise) {
                            sigma = sigma_r * d_length;
                            d_length += rand_normal((double)sigma);
                            }
#ifdef RUN_SHAD
                        if (d_length <= *z_index) {
                            *z_index = d_length;
                            *r_index = l_angle;
                            }
#endif
                        theta1 += delta_theta;
#ifdef RUN_SHAD
                        z_index++;
                        r_index++;
#endif
                        } while (theta1 <= theta2);
                    }
                address1 = address2->next_slot;
                }
            }
/*
draw horizontal lines
*/
    if (het_array[i]) {
        h_address1 = het[i];
        for (j=0;j<het_array[i];j++) {
```

```
                ftemp = sqrt( h_address1->x_min*h_address1->x_min
                                                + y_prime*y_prime + 1.);
                theta1 = (asin(h_address1->x_min / ftemp));
                ftemp = sqrt( h_address1->x_max*h_address1->x_max
                        + y_prime * y_prime + 1.);
                theta2 = (asin(h_address1->x_max / ftemp));
                phi = (las_j_pixel - i) * delta_phi;
                if (theta1 < h_angle ) {
                        if (theta1 < -h_angle)
                                theta1 = -h_angle;
                        if (theta2 > h_angle)
                                theta2 = h_angle;
                        ftemp = theta1 / delta_theta;
                        logic_flag = (sign(ftemp))+1;
                        if (logic_flag) {
                                itemp = round_pos(ftemp);
                                jindex = itemp + las_i_pixel;
                                }
                        else {
                                itemp = round_neg(ftemp);
                                jindex = itemp + las_i_pixel;
                                }
#ifdef RUN_SHAD
                        z_index =
                        depth_buf+(long)i*(long)(h_pixel+1)+(long)jindex;
                        r_index =
                        refresh_buf+(long)i*(long)(h_pixel+1)+(long)jindex;
#endif
                        do {
                                x_prime = tan(theta1) / cos(phi);
                                z_value = -planeq[3] / (planeq[0] * x_prime +
                                        planeq[1] * y_prime + planeq[2]);
                                x_value = x_prime * z_value;
                                y_value = y_prime * z_value;
                                d_length = sqrt(x_value*x_value +
                                         y_value*y_value + z_value*z_value);
                                if (add_noise) {
                                        sigma = sigma_r * d_length;
                                        d_length += rand_normal((double)sigma);
                                        }
#ifdef RUN_SHAD
                                if (d_length <= *z_index) {
                                        *z_index = d_length;
                                        *r_index = l_angle;
                                        }
#endif
```

```
                                theta1 += delta_theta;
#ifdef RUN_SHAD
                                z_index++;
                                r_index++;
#endif
                                } while (theta1 <= theta2);
                        }
                h_address1 = h_address1->next_edge;
                }
        }
/*
update active edge table
*/
        if (aet_count) {
                j_start_address = &aet;
                for (j=0;j<aet_count;j++) {
                        address1 = *j_start_address;
                        if ( (i+1) <= address1->y_max ) {
                                address1->x_min += address1->m;
                                j_start_address = &(address1->next_slot);
                                }
                        else {
                                *j_start_address = address1->next_slot;
                                delete_aet++;
                                }
                        }
                aet_count -= delete_aet;
                }
        }
skip_shad:
        return(0);
}

sort_aet(aet,aet_count)
struct et_slot  **aet;
int                             aet_count;
{
int                             sort_count,i,j,sort_flag;
struct et_slot          **i_start_address,**j_start_address;
struct et_slot          *sort_address,*next_address,*j_pre_address;

sort_count = 1;
sort_address = (*aet)->next_slot;
i_start_address = &((*aet)->next_slot);
for (i=1;i<aet_count;i++) {
        next_address = sort_address->next_slot;
```

```
        sort_count++;
        j_start_address = aet;
        sort_flag=1;
        for(j=1;j<sort_count;j++) {
                j_pre_address = *(j_start_address);
                if(sort_address->x_min < j_pre_address->x_min) {
                        sort_address->next_slot = j_pre_address;
                        *j_start_address = sort_address;
                        *i_start_address = next_address;
                        sort_flag=0;
                        j = sort_count;
                        }
                else {
                        if(sort_address->x_min == j_pre_address->x_min) {
                                if (sort_address->m < j_pre_address->m) {
                                        sort_address->next_slot = j_pre_address;
                                        *j_start_address = sort_address;
                                        *i_start_address = next_address;
                                        j = sort_count;
                                        sort_flag=0;
                                        }
                                }
                        }
                j_start_address = &(j_pre_address->next_slot);
                }
        if (sort_flag)
                i_start_address = &(sort_address->next_slot);
        sort_address = next_address;
        }
}
```

```
/*
main_sha.c --
        Dummy main program for calling the range and reflectance routine
*/

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <bscreen.h>
#include "variable.h"

#define EXEC_SHAD

int             npoly;
int             num_clip_ver;
int             h_pixel,v_pixel;
int             las_i_pixel,las_j_pixel,add_noise;
float           h_scan_angle,v_scan_angle;
float           clip_ver[MAXPVERT*2][3];
float           lambda,ar,ft,t,lambdam;

main()
{
long            array_size,i,j;
int             open(),out_ptr;
int             amb_int,itemp;
float           ftemp;
char            *out_buf;
float huge      *z_buf;
float huge      *r_buf;
float huge      *z_ptr,*r_ptr;
FILE            *fopen(),*fp;

fp = fopen("child.dat","r");
fscanf(fp,"%d %d",&h_pixel,&v_pixel);
las_i_pixel = h_pixel / 2;
las_j_pixel = v_pixel / 2;
fscanf(fp,"%f %f",&h_scan_angle,&v_scan_angle);
fscanf(fp,"%d",&amb_int);
fscanf(fp,"%d",&npoly);
fscanf(fp,"%d",&add_noise);
fscanf(fp,"%f %f %f %f %f\n",&lambda,&ar,&ft,&t,&lambdam);
```

```
#ifdef EXEC_SHAD
array_size = (h_pixel + 1 ) * (v_pixel + 1);
z_buf = (float huge *)halloc(array_size,sizeof(float));
if (z_buf == NULL) {
        printf("Insufficient memory available\n");
        exit(1);
        }
r_buf = (float huge *)halloc(array_size,sizeof(float));
if (r_buf == NULL) {
        printf("Insufficient memory available\n");
        exit(1);
        }
out_buf = malloc(h_pixel*2);
if (out_buf == NULL) {
        printf("Insufficient memory available\n");
        exit(1);
        }
for (i=0;i<array_size;i++) {
        *(z_buf+i) = 10000.;
        *(r_buf+i) = 0.;
        }
#else
array_size = 1;
z_buf = (float huge *)halloc(array_size,sizeof(float));
r_buf = (float huge *)halloc(array_size,sizeof(float));
out_buf = malloc(1);
#endif

for (i=0;i<npoly;i++) {
        sccurset(20,15);
        printf("    ");
        sccurset(20,15);
        printf("%3d",npoly-i);
        fscanf(fp,"%d",&num_clip_ver);
        for (j=0;j<num_clip_ver;j++)
                fscanf(fp,"%f %f %f",&(clip_ver[j][0]),&(clip_ver[j][1]),
                                                        &(clip_ver[j][2]));
        if (num_clip_ver >= 3)
                shad_depth(r_buf,z_buf);
        }
sccurset(20,15);
printf("    \n");
fclose(fp);
out_ptr = creat("frame.out",S_IREAD|S_IWRITE);
if (out_ptr == -1) {
        printf("open failed on output file\n");
```

```
        exit(1);
        }
setmode(out_ptr,O_BINARY);
for (i=0; i<v_pixel;i++) {
        z_ptr = z_buf + i * (h_pixel + 1);
        r_ptr = r_buf + i * (h_pixel + 1);
        for (j=0;j<h_pixel;j++) {
                ftemp = *(r_ptr+j);
                ftemp = ftemp * 255 + 0.5;
                itemp = (int) ftemp;
                *(out_buf+j*2) = itemp;
                ftemp = *(z_ptr+j);
                if (fabs(ftemp-10000) <= EPSILON)
                        *(out_buf+j*2+1) = 0;
                else {
                        ftemp = fmod(ftemp,amb_int);
                        ftemp = (ftemp * 255 / amb_int) + 0.5;
                        itemp = (int) ftemp;
                        *(out_buf+j*2+1) = itemp;
                        }
                }
        write(out_ptr,out_buf,(unsigned)h_pixel*2);
        }
close(out_ptr);
}
```

```
/*
spath.c --
        Minimum distance collision-free path planner using the SDSA algorithm

        Written by Chris K. Wu, Jun 1989
*/

#include <stdio.h>
#include <bscreen.h>
#include <bgraph.h>
#include <conio.h>
#include <malloc.h>
#include <string.h>
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <stdlib.h>
#include <math.h>
#define     WRTMODE         CHARS_ONLY+MOVE_CUR+CUR_AFTER
#define     BLACK           0
#define     BLUE            1
#define     GREEN           2
#define     CYAN            3
#define     RED             4
#define     MAGENTA         5
#define     BROWN           6
#define     WHITE           7
#define     GRAY            8
#define     LBLUE           9
#define     LGREEN          10
#define     LCYAN           11
#define     LRED            12
#define     LMAGENTA        13
#define     LYELLOW         14
#define     LWHITE          15
#define     W               3
#define     SCALE           30.6705
typedef     struct {
                int screenMask[16],
                cursorMask[16],
                hotX,
                hotY;
                } gCursor_type;
```

```
gCursor_type graphicCursor = {
        {65535,65535,65535,65535,65535,65535,65535,65535,
        65535,65535,65535,65535,65535,65535,65535,65535},
        {49152,61440,32640,32320,15648,15504,8776,4388,2194,
        1097,549,275,143,124,0,0}
        };

main()
{
unsigned        d,noOfButtons;
int             gmode,resp;
int             status,c;
int             quit = 0,first_time=1,newmap,clearflag;
int             horiz, vert, horizMotion, vertMotion;
int             buttonStatus, buttonPressCount, button;
float huge      *map;
float           me;
long            i,array_size;

array_size=128*128;
map = (float huge *)halloc(array_size,sizeof(float));
for (i=0;i<array_size;i++) map[i] = 0.0;
gmode = 16;
grinit(gmode,0,0);
draw_color_code();
drawbox();
main_menu();
FlagReset(&status, &noOfButtons);
SetGraphicCursor (&graphicCursor);
SetMickeysPerPixel (8,8);
SetCursorPos (550,100);
ShowCursor();
for ( ; quit != 1 ; ) {
        button = 0;  /* left button */
        GetPosBut (&buttonStatus, &horiz, &vert);
        c = kbhit();
        if ((buttonStatus+c) != 0) {
                if ( c != 0 ) {
                        c = getch();
                        switch (c) {
                                case 'B':
                                case 'b':
                                        c = 0;
                                        break;
                                case 'L':
                                case 'l':
```

```
                        c = 1;
                        break;
                case 'S':
                case 's':
                        c = 2;
                        break;
                case 'P':
                case 'p':
                        c = 3;
                        break;
                case 'R':
                case 'r':
                        c = 5;
                        break;
                case 'E':
                case 'e':
                        c = 7;
                        break;
                default:
                        c = 8;
                }
        }
else {
        if ( (horiz >= 472) && (horiz <= 587) ) {
                if ( (vert >= 64) && (vert <= 286) )
                        c = (vert - 63 ) / 28;
                else c = 8;
                }
        else c = 8;
        }
switch (c) {
        case 0:
                edit_or_new(&newmap);
                if (newmap) {
                        sure_yn(&resp);
                        if (resp == 1) {
                                me = 0;
                                if (first_time != 1) {
                                        clearmap();
                                        drawbox();
                                        }
                                else
                                        first_time = 0;
                                }
                        }
                if ((newmap == 0) || (resp == 1))
```

```
                        buildmap(&me,map,newmap);
                    break;
                case 1:
                    sure_yn(&resp);
                    if (resp == 1) {
                        if (first_time != 1)
                            clearflag=1;
                        else {
                            clearflag = 0;
                            first_time = 0;
                            }
                        load_map(&me,map);
                        sccurset(4,41);
                        printf("%7.1f",me);
                        drawmap(me,map,clearflag);
                        }
                    break;
                case 2:
                    save_map(&me,map);
                    break;
                case 3:
                    break;
                case 4:
                    break;
                case 5:
                    findpath(map);
                    break;
                case 6:
                    break;
                case 7:
                    quit = 1;
                    break;
                default:
                    ;
                }
            }
        }
FlagReset(&status, &noOfButtons);
gmode = 0;
grinit(gmode,0,0);
scpclr();
exit(0);

}
```

```
main_menu()
{
int i,j,c,row,col;
PT startpt,endpt;

sccurset(3,58);
printf("3-D MAP BUILDER");
sccurset(5,59);
c = 0x42;
scattrib(LGREEN,BLACK,c,1);
sccurset(5,60);
printf("uild map");
sccurset(7,59);
c = 0x4c;
scattrib(LGREEN,BLACK,c,1);
sccurset(7,60);
printf("oad map");
sccurset(9,59);
c = 0x53;
scattrib(LGREEN,BLACK,c,1);
sccurset(9,60);
printf("ave map");
sccurset(11,59);
printf("set ");
sccurset(11,63);
c = 0x50;
scattrib(LGREEN,BLACK,c,1);
sccurset(11,64);
printf("arameters");
sccurset(15,59);
c = 0x52;
scattrib(LGREEN,BLACK,c,1);
sccurset(15,60);
printf("un");
sccurset(19,59);
c = 0x45;
scattrib(LGREEN,BLACK,c,1);
sccurset(19,60);
printf("xit");

/*draw vertital box line*/
sccurset(4,58);
c = 0xc9;
printf("%c",c);
row = 5;
for(i=0;i<7;i++) {
```

```
        sccurset(row++,58);
        c = 0xba;
        printf("%c",c);
        sccurset(row++,58);
        c = 0xc7;
        printf("%c",c);
        }
sccurset(row++,58);
c = 0xba;
printf("%c",c);
sccurset(row,58);
c = 0xc8;
printf("%c",c);

sccurset(4,73);
c = 0xbb;
printf("%c",c);
row = 5;
for(i=0;i<7;i++) {
        sccurset(row++,73);
        c = 0xba;
        printf("%c",c);
        sccurset(row++,73);
        c = 0xb6;
        printf("%c",c);
        }
sccurset(row++,73);
c = 0xba;
printf("%c",c);
sccurset(row,73);
c = 0xbc;
printf("%c",c);
/*draw horizontal box line*/
col = 59;
c = 0xcd;
for (i=0;i<14;i++) {
        sccurset(4,col);
        printf("%c",c);
        sccurset(20,col++);
        printf("%c",c);
        }
row = 6;
c = 0xc4;
for (i=0;i<7;i++) {
        row = 6 + 2*i;
        col = 59;
```

```
        for (j=0;j<14;j++) {
                sccurset(row,col++);
                printf("%c",c);
                }
        }
sccurset(22,60);
printf("DESIGNED BY");
sccurset(23,60);
printf("CHRIS K. WU");
sccurset(24,59);
printf("(713) 668-6102");

}


draw_color_code()
{
int i,j,length;
PT startpt,endpt;

scwrbuf(0,14,0,
        "3-D NAVIGATION DEMO PROGRAM",LGREEN,BLACK,WRTMODE);
sccurset(1,18);
printf("ELEVATION COLOR CODE");

for (i=0;i<14;i++) {
        startpt.y = endpt.y = 28 + i;
        length = 19;
        startpt.x = 40;
        endpt.x = startpt.x + length;
        grline(&startpt,&endpt,8);
        length = 23;
        for (j=5;j>0;j--) {
                startpt.x = endpt.x+1;
                endpt.x = startpt.x + length;
                grline(&startpt,&endpt,j);
                }
        length = 15;
        startpt.x = endpt.x+1;
        endpt.x = startpt.x + length;
        grline(&startpt,&endpt,0);
        length = 23;
        for (j=9;j<14;j++) {
                startpt.x = endpt.x+1;
                endpt.x = startpt.x + length;
                grline(&startpt,&endpt,j);
```

```
                }
        length = 19;
        startpt.x = endpt.x+1;
        endpt.x = startpt.x + length;
        grline(&startpt,&endpt,14);
        }

sccurset(3,5);
printf("-1 .8 .6 .4 .2 .1 0 .1 .2 .4 .6 .8 +1 x M.E.");
sccurset(4,14);
printf("MAXIMUM ELEVATION (M.E.) = 0");
startpt.y = endpt.y = 14 + i;
length = 19;
startpt.x = 344;
endpt.x = startpt.x + length;
for (i=0;i<10;i++) {
        startpt.y = endpt.y = 16 + i;
        grline(&startpt,&endpt,7);
        }
sccurset(1,46);
printf("UNSCANNED");
for (i=0;i<10;i++) {
        startpt.y = endpt.y = 30 + i;
        grline(&startpt,&endpt,6);
        }
sccurset(2,46);
printf("OCCLUDED");
}


drawbox()
{
int i;
PT startpt,endpt;

startpt.y = 70;
startpt.x = endpt.x = 28;
endpt.y = 334;
grline(&startpt,&endpt,WHITE);

startpt.y = endpt.y = 72;
startpt.x = 25;
endpt.x = 413;
grline(&startpt,&endpt,WHITE);

startpt.y = endpt.y = 329;
```

```
startpt.x = 25;
endpt.x = 415;
grline(&startpt,&endpt,WHITE);

startpt.y = 72;
startpt.x = endpt.x = 413;
endpt.y = 334;
grline(&startpt,&endpt,WHITE);
/*draw horizontal scale line*/
startpt.x = 25;
endpt.x = 27;
startpt.y = endpt.y = 129;
for (i=0; i<5; i++) {
        grline(&startpt,&endpt,WHITE);
        startpt.y = endpt.y = endpt.y + 40;
        }
sccurset(5,0);
printf("127");
sccurset(8,0);
printf("100");
sccurset(11,1);
printf("80");
sccurset(14,1);
printf("60");
sccurset(17,1);
printf("40");
sccurset(20,1);
printf("20");
sccurset(23,2);
printf("0");
sccurset(4,3);
printf("Y");
/*draw vertical scale line*/
startpt.y = 330;
endpt.y = 334;
startpt.x = endpt.x = 328;
for (i=0; i<5; i++) {
        grline(&startpt,&endpt,WHITE);
        startpt.x = endpt.x = endpt.x - 60;
        }
sccurset(24,3);
printf("0");
sccurset(24,10);
printf("20");
sccurset(24,17);
printf("40");
```

```
sccurset(24,25);
printf("60");
sccurset(24,32);
printf("80");
sccurset(24,40);
printf("100");
sccurset(24,49);
printf("127");
sccurset(23,52);
printf("X");
}


sure_yn(resp)
int *resp;
{
unsigned noOfButtons;
int i,quit;
int c;
int horiz, vert;
int buttonStatus;
HideCursor();
/* Draw box */
scbox(8,57,14,74,15,0,LBLUE);
/*clear box*/
for (i=9;i<14;i++) {
        sccurset(i,58);
        printf("                  ");
        }
sccurset(10,59);
printf("Are you sure?");
sccurset(12,62);
printf("(Y / N)");
ShowCursor();
quit = 0;
for ( ; quit != 1 ; ) {
        GetPosBut (&buttonStatus, &horiz, &vert);
        c = kbhit();
        if ((buttonStatus+c) != 0) {
                if ( c != 0 ) {
                        c = getch();
                        switch (c) {
                                case 'N':
                                case 'n':
                                        *resp = 0;
                                        quit = 1;
```

```
                        break;
                    case 'Y':
                    case 'y':
                        *resp = 1;
                        quit = 1;
                        break;
                    default:
                        ;
                    }
                }
            else {
                if ( (horiz >= 496) && (horiz <= 543) ) {
                    if ( (vert >= 168) && (vert <= 181) ) {
                        if ( horiz <= 519 ) *resp = 1;
                        else *resp = 0;
                        quit = 1;
                        }
                    }
                }
            }
    if (*resp == 0) {
        /*clear window*/
        HideCursor();
        for (i=8;i<15;i++) {
            sccurset(i,57);
            printf("                    ");
            }
        main_menu();
        ShowCursor();
        }
}


edit_or_new(resp)
int *resp;
{
unsigned noOfButtons;
int i,quit;
int c,col;
int horiz, vert;
int buttonStatus;
HideCursor();
/* Draw box */
scbox(8,57,14,74,15,0,LBLUE);
/*clear box*/
```

```
for (i=9;i<14;i++) {
        sccurset(i,58);
        printf("                    ");
        }
sccurset(10,58);
c = 0x31;
scattrib(LGREEN,BLACK,c,1);
sccurset(10,60);
printf("New map");
c = 0xc4;
for(col=58;col<74;col++) {
        sccurset(11,col);
        printf("%c",c);
        }
sccurset(12,58);
c = 0x32;
scattrib(LGREEN,BLACK,c,1);
sccurset(12,60);
printf("Edit map");
ShowCursor();
quit = 0;
for ( ; quit != 1 ; ) {
        GetPosBut (&buttonStatus, &horiz, &vert);
        c = kbhit();
        if ((buttonStatus+c) != 0) {
                if ( c != 0 ) {
                        c = getch();
                        switch (c) {
                                case '1':
                                        *resp = 1;
                                        quit = 1;
                                        break;
                                case '2':
                                        *resp = 0;
                                        quit = 1;
                                        break;
                                default:
                                        ;
                                }
                        }
                else {
                        if ( (horiz >= 460) && (horiz <= 596) ) {
                                if ( (vert >= 133) && (vert <= 189) ) {
                                        if ( vert >= 161 ) *resp = 0;
                                        else *resp = 1;
                                        quit = 1;
```

```
                                }
                            }
                        }
                    }
                }
            }
        }

save_map(me,map)
float *me;
float huge *map;
{
char buf[15],out_file[15];
int i,fp1;
float huge *pt;
unsigned int size;

HideCursor();
/* Draw box */
scbox(8,57,14,74,15,0,LBLUE);
/*clear box*/
for (i=9;i<14;i++) {
        sccurset(i,58);
        printf("                  ");
        }
sccurset(10,59);
printf("Enter filename");
sccurset(12,59);
printf(":");
get_resp(12,61,buf);
i=strcspn(buf,".");
if (i==0) i = strlen(buf);
strncpy(out_file,buf,i);
out_file[i]=0;
strcat(out_file,".map");
fp1=creat(out_file,S_IREAD|S_IWRITE);
setmode(fp1,O_BINARY);
write(fp1,(char *)me,sizeof(float));
size = 128*64*sizeof(float);
write(fp1,(char *)map,size);
i=128*64;
pt = &(map[i]);
write(fp1,(char *)pt,size);
close(fp1);
/*clear window*/
for (i=8;i<15;i++) {
        sccurset(i,57);
```

```
        printf("                    ");
        }
main_menu();
ShowCursor();
}

load_map(me,map)
float *me;
float huge *map;
{
char buf[15],in_file[15];
int i,fp1;
float huge *pt;
unsigned int size;

HideCursor();
/* Draw box */
scbox(8,57,14,74,15,0,LBLUE);
/*clear box*/
for (i=9;i<14;i++) {
        sccurset(i,58);
        printf("                    ");
        }
sccurset(10,59);
printf("Enter filename");
sccurset(12,59);
printf(":");
get_resp(12,61,buf);
i=strcspn(buf,".");
if (i==0) i = strlen(buf);
strncpy(in_file,buf,i);
in_file[i]=0;
strcat(in_file,".map");
fp1=open(in_file,O_RDONLY);
setmode(fp1,O_BINARY);
read(fp1,(char *)me,sizeof(float));
size = 128*64*sizeof(float);
read(fp1,(char *)map,size);
i=128*64;
pt = &(map[i]);
read(fp1,(char *)pt,size);
close(fp1);
/*clear window*/
for (i=8;i<15;i++) {
        sccurset(i,57);
        printf("                    ");
```

```
        }
main_menu();
ShowCursor();
}

findpath(map)
float huge *map;
{
int i,quit;
int c;
int horiz, vert;
int buttonStatus;
char buf[80];
float xs,ys,xe,ye,a[3];

HideCursor();
/* Draw box */
scbox(7,52,17,79,15,0,LYELLOW);
/*clear box*/
for(i=8;i<17;i++) {
        sccurset(i,53);
        printf("                            ");
        }
/*sccurset(8,53);
printf("enter the starting point");
quit=0;
ShowCursor();
for ( ; quit != 1 ; ) {
        GetPosBut (&buttonStatus, &horiz, &vert);
        if (buttonStatus != 0) {
                if ( (horiz >= 29) && (horiz <= 412) ) {
                        if ( (vert >= 73) && (vert <= 328) ) {
                                xs = (horiz - 29) / 3;
                                ys = 127 - ((vert - 73) / 2);
                                quit = 1;
                                }
                        }
                }
        }
HideCursor();*/
xs = 63;
ys = 0;
drawpoint((long)(127.-ys),(long)xs,LWHITE);
drawpoint((long)(128.-ys),(long)xs,LWHITE);
drawpoint((long)(127.-ys),(long)xs+1,LWHITE);
drawpoint((long)(128.-ys),(long)xs+1,LWHITE);
```

```
sccurset(8,53);
printf("the starting point");
sccurset(9,53);
printf("xs=%5.1f, ys=%5.1f",xs,ys);
sccurset(10,53);
printf("enter the end point");
quit=0;
ShowCursor();
/*while (buttonStatus != 0)
        GetPosBut (&buttonStatus, &horiz, &vert);*/
for ( ; quit != 1 ; ) {
        GetPosBut (&buttonStatus, &horiz, &vert);
        if (buttonStatus != 0) {
                if ( (horiz >= 29) && (horiz <= 412) ) {
                        if ( (vert >= 73) && (vert <= 328) ) {
                                xe = (horiz - 29) / 3;
                                ye = 127 - ((vert - 73) / 2);
                                quit = 1;
                                }
                        }
                }
        }
HideCursor();
drawpoint((long)(127.-ye),(long)xe,LWHITE);
drawpoint((long)(128.-ye),(long)xe,LWHITE);
drawpoint((long)(127.-ye),(long)xe+1,LWHITE);
drawpoint((long)(128.-ye),(long)xe+1,LWHITE);
sccurset(11,53);
printf("xe=%5.1f, ye=%5.1f",xe,ye);
sccurset(14,53);
printf("searching for the path ...");
/*save_dat(&xs,&ys,&xe,&ye,map);*/
searchpath(xs,ys,xe,ye,map,a);
sccurset(14,53);
printf("                              ");
sccurset(12,53);
printf("best path found:");
sccurset(14,53);
printf("x =%5dt^2+%5dt+%5d",(int)a[0],(int)(xe-xs-a[0]),(int)xs);
sccurset(15,53);
printf("y =%5dt^2+%5dt+%5d",(int)a[1],(int)(ye-ys-a[1]),(int)ys);
drawpath(xs,ys,xe,ye,a);
sccurset(16,53);
printf("press any key to continue");
quit = 0;
for ( ; quit != 1 ; ) {
```

```
        GetPosBut (&buttonStatus, &horiz, &vert);
        c = kbhit();
        if ((buttonStatus+c) != 0)
                quit = 1;
        }
for(i=7;i<18;i++) {
        sccurset(i,52);
        printf("                                ");
        }
main_menu();
ShowCursor();
}

drawpath(xs,ys,xe,ye,x)
float xs,ys,xe,ye,*x;
{
int i,step=80;
long xi,yi;
float a1,b1,c1,a2,b2,c2,d2,t;
float dx,dy,l;
FILE *fp, *fopen();

a1 = x[0];
b1 = xe - xs - a1;
c1 = xs;
a2 = x[1];
b2 = ye - ys - a2;
c2 = ys;
/*save path*/
fp = fopen ("path.dat","w");
fprintf(fp,"%f %f 0.0\n", a1*SCALE, b1*SCALE);
fprintf(fp,"%f %f 0.0\n", a2*SCALE, b2*SCALE);
fclose(fp);

for(i=0; i<step; i++) {
        t = (float)i / (float)step;
        dx = (2.*a1*t + b1);
        dy = (2.*a2*t + b2);
        l = (float)sqrt(dx*dx+dy*dy) / W;
        dx /= l;
        dy /= l;
        xi = (long)(a1*t*t+b1*t+c1 - dy + 0.5) ;
        yi = (long)(a2*t*t+b2*t+c2 + dx +0.5);
        if (yi >= 0 )
                drawpoint((127-yi),xi,LWHITE);
        xi = xi + 2.*dy;
```

```
        yi = yi - 2.*dx;
        if (yi >= 0 )
                drawpoint((127-yi),xi,LWHITE);
        }
}

save_dat(xs,ys,xe,ye,map)
float *xs,*ys,*xe,*ye;
float huge *map;
{
char out_file[15];
int i,fp1;
float huge *pt;
unsigned int size;

sprintf(out_file,"navigate.dat");
fp1=creat(out_file,S_IREAD|S_IWRITE);
setmode(fp1,O_BINARY);
write(fp1,(char *)xs,sizeof(float));
write(fp1,(char *)ys,sizeof(float));
write(fp1,(char *)xe,sizeof(float));
write(fp1,(char *)ye,sizeof(float));
size = 128*64*sizeof(float);
write(fp1,(char *)map,size);
i=128*64;
pt = &(map[i]);
write(fp1,(char *)pt,size);
close(fp1);
}
```

```
/*
navigate.c --
        path searching subroutine for robots with 2W span distance
*/

#include <stdio.h>
#include <malloc.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <math.h>

#define SMLNUM    1e-4
#define BIGNUM    1e8
#define XMIN      0.
#define XMAX      127.
#define YMIN      0.
#define YMAX      127.
#define W         3;
float huge        *map;
float             xs,ys,xe,ye;

searchpath(pxs,pys,pxe,pye,pmap,a)
float huge    *pmap;
float         pxs,pys,pxe,pye;
float         *a;
{

char          in_file[15];
int           i,j,fp1,nfunk,ndim;
unsigned int  size;
long          array_size;
float huge    *pt;
double        funkpt(),funktk();
double        p[4][3],y[4],q[4][3],ftol,l,ysave,c;

xs=pxs;ys=pys;
xe=pxe;ye=pye;
map = pmap;
ndim = 2;
ftol = .01;
/*first vertex of the starting simplex, a straight line path*/
p[1][1] = p[1][2]= 0.;
/*second vertex of the starting simplex, a quadratic curve path*/
p[2][1] = 100.; p[2][2] = -75.;
```

```
/*third vertex of the starting simplex, a cubic curve path*/
p[3][1] = 200.; p[3][2] = -150.;

nmsimplex(p,y,ndim,ftol,&nfunk,funkpt);
q[1][1] = -50.; q[1][2]= 0.;
q[2][1] = -125.; q[2][2] = -75.;
q[3][1] = -200.; q[3][2] = -150.;

nmsimplex(q,y,ndim,ftol,&nfunk,funkpt);
p[1][1] = p[1][2]= 0.;
p[3][1] = q[3][1]; p[3][2] = q[3][2];
nmsimplex(p,y,ndim,ftol,&nfunk,funkpt);
q[1][1] = p[1][1];
q[1][2] = p[1][2] + 20;
q[2][1] = p[1][1] - 20;
q[2][2] = q[3][2] = p[1][2] - 20;
q[3][1] = p[1][1] + 20;
nmsimplex(q,y,ndim,ftol,&nfunk,funktk);

for (i=0;i<ndim;i++) a[i] = (float) q[2][i+1];
}

#define        STEP  40

double funkpt(x)
double *x;
{
int            i,di,l_flag,flag;
long           xlow,xhigh,ylow,yhigh;
float          rx[STEP],ry[STEP],rz[STEP],frac,h1,h2,t;
double         a1,b1,c1,a2,b2,c2,d2,xmax,xmin,ymax,ymin,num,den,l,dx,dy,dz;
double         tmax,tmin;

l_flag = 1;
a1 = x[1];
c1 = xs;
b1 = xe - xs - a1;
/*check if xmax is within allowed region*/
if ( fabs(a1) > SMLNUM) {
        tmax = -b1/(2.*a1);
        if ( (tmax >= 0.) || (tmax <= 1.)  ) {
                xmax = a1*tmax*tmax+b1*tmax+c1;
                if ((xmax > XMAX) || ( xmax < XMIN) ) {
                        l = BIGNUM;
                        l_flag = 0;
                        }
```

```
                }
        }
a2 = 0.;
b2 = x[2];
d2 = ys;
c2 = ye - ys - a2 - b2;
/*check if ymax and ymin are within allowed region*/
if ( (flag = (fabs(a2) > SMLNUM)) || (fabs(b2) > SMLNUM) ) {
        if ( flag ) {
                num = 4.*b2*b2 - 12.*a2*c2;
                if ( num >= 0.) {
                        num = sqrt(num);
                        den = 6 * a2;
                        tmax = (-2.*b2 + num)/den;
                        tmin = (-2.*b2 - num)/den;
                        if ( (tmax >= 0.) || (tmax <= 1.)  ) {
                                ymax =
                                a2*tmax*tmax*tmax+b2*tmax*tmax+c2*tmax+d2;
                                if ((ymax > YMAX) || ( ymax < YMIN) ) {
                                        l = BIGNUM;
                                        l_flag = 0;
                                        }
                                }
                        if ( (tmin >= 0.) || (tmin <= 1.)  ) {
                                ymin = a2*tmin*tmin*tmin+b2*tmin*tmin+c2*tmin+d2;
                                if ((ymin > YMAX) || ( ymin < YMIN) ) {
                                        l = BIGNUM;
                                        l_flag = 0;
                                        }
                                }
                        }
                }
        else {
                tmax = -c2/(2.*b2);
                if ( (tmax >= 0.) || (tmax <= 1.)  ) {
                        ymax = b2*tmax*tmax+c2*tmax+d2;
                        if ((ymax > XMAX) || ( ymax < XMIN) ) {
                                l = BIGNUM;
                                l_flag = 0;
                                }
                        }
                }
        }
if (l_flag) {
        rx[0] = xs;
        ry[0] = ys;
```

```
        rz[0] = map[(long)(127 - ys - 1)*128+(long)xs]; /* ys - 1 is used
                                because last row of data is incorrect */
        di = STEP-1;
        for(i=1;i<di;i++) {
                t = (float)i/(float)di;
                rx[i] = (float)(a1*t*t+b1*t+c1);
                ry[i] = (float)(a2*t*t*t+b2*t*t+c2*t+d2);
                xlow = (long)rx[i];
                xhigh = xlow + 1;
                ylow = 127 - (long) ry[i];
                yhigh = ylow - 1;
                frac = ry[i] - (int)ry[i];
                h1 = map[ylow*128+(long)xlow]
                +frac*(map[yhigh*128+(long)xlow]-map[ylow*128+(long)xlow]);
                h2 = map[ylow*128+(long)xhigh]
                +frac*(map[yhigh*128+(long)xhigh]-map[ylow*128+(long)xhigh]);
                frac = rx[i] - xlow;
                rz[i] = h1 + frac * (h2-h1);
                }
        rx[di] = xe;
        ry[di] = ye;
        rz[di] = map[ (127-(long)ye) *128+(long)xe];
        l = 0.;
        /* check if there is any noisy data along the path */
        for (i=1;i<STEP-1;i++)
                if ( rz[i] < -400 )
                        if (rz[i-1]+rz[i+1] > -400) rz[i] = (rz[i-1]+rz[i+1])/2;
        for(i=1;i<STEP;i++) {
                dx = rx[i]-rx[i-1];
                dy = ry[i]-ry[i-1];
                dz = rz[i]-rz[i-1];
                l += sqrt(dx*dx+dy*dy+dz*dz);
                }
        }
return(l);
}

double funktk(x)
double          *x;
{
int             i,di,l_flag,flag;
long            xlow,xhigh,ylow,yhigh;
float           rrx[STEP],rry[STEP],rrz[STEP],frac,h1,h2,t;
float           lrx[STEP],lry[STEP],lrz[STEP],zs;
double          a1,b1,c1,a2,b2,c2,d2,xmax,xmin,ymax,ymin,num,den,l,dx,dy,dz;
double          tmax,tmin,norm;
```

```
l_flag = 1;
a1 = x[1];
c1 = xs;
b1 = xe - xs - a1;
/*check if xmax is within allowed region*/
/*if ( fabs(a1) > SMLNUM) {
        tmax = -b1/(2.*a1);
        if ( (tmax >= 0.) || (tmax <= 1.)  ) {
                xmax = a1*tmax*tmax+b1*tmax+c1;
                if ((xmax > XMAX) || ( xmax < XMIN) ) {
                        l = BIGNUM;
                        l_flag = 0;
                        }
                }
        } */
a2 = x[2];
c2 = ys;
b2 = ye - ys - a2;
/*if (l_flag) {*/
di = STEP-1;
zs = map[(long)(127 - ys - 1)*128+(long)xs]; /*ys-1 is used because last row
                                                of data is incorrect */
/* check the left and right tracks */
        for(i=0;i<STEP;i++) {
                t = (float)i /(float)di;
                dx = (2.*a1*t + b1);
                dy = (2.*a2*t + b2);
                norm = sqrt(dx*dx + dy*dy) / W;
                dx /= norm;
                dy /= norm;
                lrx[i] = (float)(a1*t*t+b1*t+c1) - dy;
                lry[i] = (float)(a2*t*t+b2*t+c2) + dx;
                rrx[i] = lrx[i] + 2.*dy;
                rry[i] = lry[i] - 2.*dx;
                if (lry[i] < 1) lrz[i] = zs;
                else {
                        xlow = (long)lrx[i];
                        xhigh = xlow + 1;
                        ylow = 127 - (long) lry[i];
                        yhigh = ylow - 1;
                        frac = lry[i] - (int)lry[i];
                        h1 = map[ylow*128+(long)xlow] + frac *
                        (map[yhigh*128+(long)xlow]-map[ylow*128+(long)xlow]);
                        h2 = map[ylow*128+(long)xhigh] + frac *
                        (map[yhigh*128+(long)xhigh]-map[ylow*128+(long)xhigh]);
                        frac = lrx[i] - xlow;
```

```
                lrz[i] = h1 + frac * (h2-h1);
                }
        if (rry[i] < 1) rrz[i] = zs;
        else {
                xlow = (long)rrx[i];
                xhigh = xlow + 1;
                ylow = 127 - (long) rry[i];
                yhigh = ylow - 1;
                frac = rry[i] - (int)rry[i];
                h1 = map[ylow*128+(long)xlow] + frac *
                (map[yhigh*128+(long)xlow]-map[ylow*128+(long)xlow]);
                h2 = map[ylow*128+(long)xhigh] + frac *
                (map[yhigh*128+(long)xhigh]-map[ylow*128+(long)xhigh]);
                frac = rrx[i] - xlow;
                rrz[i] = h1 + frac * (h2-h1);
                }
        }
l = 0.;
/* check if there is any noisy data along the path */
for (i=1;i<STEP-1;i++)
        if ( lrz[i] < -400. )
                if (lrz[i-1]+lrz[i+1] > -400.) lrz[i] = (lrz[i-1]+lrz[i+1])/2;
for (i=1;i<STEP-1;i++)
        if ( rrz[i] < -400. )
                if (rrz[i-1]+rrz[i+1] > -400.) rrz[i] = (rrz[i-1]+rrz[i+1])/2;
/* check for obstacles */
for(i=1;i<STEP;i++) {
        dx = lrx[i]-lrx[i-1];
        dy = lry[i]-lry[i-1];
        dz = lrz[i]-lrz[i-1];
        l += sqrt(dx*dx+dy*dy+dz*dz);
        dx = rrx[i]-rrx[i-1];
        dy = rry[i]-rry[i-1];
        dz = rrz[i]-rrz[i-1];
        l += sqrt(dx*dx+dy*dy+dz*dz);
        }
return(l);
}
```

```
/* NMSIMPLEX.C : Downhill Nelder-Mead simplex searching algorithm
        INPUT
                ndim    -       dimension of the parameter space
                p       -       matrix p[ndim+1][ndim] stores the vertices of the
                                starting simplex. Its ndim+1 rows are ndim vectors
                                which are the vertices of the starting simplex
                funk    -       multidimensional function funk(x)
                ftol    -       the fractional convergence tolerance to be achieved in
                                the functional value.
        OUTPUT
                p       -       new ndim+1 points all within ftol of a minimum
                                function value
                y       -       vector y[dim+1] whose elements are the values of funk
                                evaluated at the ndim+1 vertices of p
                nfunk   -       number of function evaluation taken.
*/

#define         NMAX            5000
#define         ALPHA           1.0
#define         BETA            0.5
#define         GAMMA           3.0
#define         ZETA            2.5

#define GET_PSUM for (j=1;j<=ndim;j++) { for (i=1, sum=0.0;i<=mpts;i++)\
                        sum+= p[i*(ndim+1)+j]; psum[j]=sum;}

nmsimplex(p,y,ndim,ftol,nfunk,funk)
double          *p,*y,ftol;
int             ndim, *nfunk;
double          (*funk)();
{
int             i,j,ilo,ihi,inhi,mpts=ndim+1;
double          ytry,ysave,sum,rtol,amotry(),*floatpt,*psum;

psum=(double *)malloc(mpts*sizeof(double));

/*
initialize the funk value of the starting simplex
*/
for(i=1;i<=ndim+1;i++) {
        floatpt = &(p[i*mpts]);
        y[i] = (*funk)(floatpt);
        }
*nfunk=ndim+1;
```

```
GET_PSUM
for (;;){
        ilo=1;

/*
get the lowest and second highest and highest points
*/
        ihi =y[1]>y[2] ? (inhi=2,1):(inhi=1,2);
        for (i=1;i<=mpts;i++) {
                if (y[i] < y[ilo]) ilo = i;
                if (y[i] > y[ihi]) {
                        inhi=ihi;
                        ihi=i;
                        }
                else if (y[i]> y[inhi])
                        if (i != ihi) inhi=i;
                }

/*
check for convergence
*/
        rtol=fabs(y[ihi]-y[ilo]);
        if (rtol <= ftol) break;
        if (*nfunk >=NMAX) {
                printf("Too many iterations in AMOEBA\n");
                exit(1);
                }

/*
find the funk value at the reflection point
*/
        ytry=amotry(p,y,psum,ndim,ihi,nfunk,funk,-ALPHA);
        if (ytry <= y[ilo]) /* check if it is a sucessful reflection */

/*
find the funk value at the extend reflection point
*/
                ytry=amotry(p,y,psum,ndim,ihi,nfunk,funk,GAMMA);

        else if (ytry >= y[inhi]) {

/*
the reflected point is worse than the second highest point
*/
                ysave = y[ihi];
```

```
/*
perform a one dimensional contraction
*/

                ytry=amotry(p,y,psum,ndim,ihi,nfunk,funk,BETA);
                if (ytry >= ysave) {
/*
situation can't be improved
*/
                        for (i=1;i<=mpts;i++) {
                                if (i != ilo) {
/*
contract arround the lowest point
*/
                                        for (j=1;j<=ndim;j++) {
                                                psum[j]=0.5*(p[i*mpts+j]+p[ilo*mpts+j]);
                                                p[i*mpts+j]=psum[j];
                                                }
                                        y[i]=(*funk)(psum);
                                        }
                        }
                        *nfunk += ndim;
                        GET_PSUM
                        }
                }
        }
free((char *)psum);
}

double amotry(p,y,psum,ndim,ihi,nfunk,funk,fac)
double          *p,*y,*psum,fac;
int             ndim,ihi,*nfunk;
double          (*funk)();
{
int             j,mpts=ndim+1;
double          fac1,fac2,ytry,*ptry;

ptry=(double *)malloc(mpts*sizeof(double));
fac1=(1.0-fac)/ndim;
fac2=fac1-fac;
for (j=1;j<=ndim;j++) ptry[j]=psum[j]*fac1-p[ihi*mpts+j]*fac2;
ytry=(*funk)(ptry);
++(*nfunk);
if ( (ytry-y[ihi]) < -SMLNUM ) {
        y[ihi]=ytry;
                for (j=1;j<=ndim;j++){
```

```
                        psum[j] += ptry[j]-p[ihi*mpts+j];
                        p[ihi*mpts+j]=ptry[j];
                }
        }
free((char *)ptry);
return(ytry);
}
```

```
/*
buildmap.c --
        elevation map plotting subroutine
*/

#include <stdio.h>
#include <bscreen.h>
#include <bgraph.h>
#include <conio.h>
#include <malloc.h>
#include <bkeybd.h>
#include <string.h>
#include <bapplic.h>
#include <math.h>
#define         WRTMODE         CHARS_ONLY+MOVE_CUR+CUR_AFTER
#define         BLACK           0
#define         BLUE            1
#define         GREEN           2
#define         CYAN            3
#define         RED             4
#define         MAGENTA         5
#define         BROWN           6
#define         WHITE           7
#define         GRAY            8
#define         LBLUE           9
#define         LGREEN          10
#define         LCYAN           11
#define         LRED            12
#define         LMAGENTA        13
#define         LYELLOW         14
#define         LWHITE          15
#define         PI              3.14159
#define         UNSCANNED       -1999.0
#define         OCCLUDED        -1998.0

buildmap(climax,map,newmap)
float           *climax;
float huge      *map;
int             newmap;
{
unsigned        noOfButtons;
int     i,quit=0;
int     c,flag=0;
int     horiz, vert;
int     buttonStatus;
long    row,col;
```

```
HideCursor();
/* clear array */
if (newmap) {
        for (row=0;row<128; row++)
                for (col=0;col<128;col++) map[col+row*128] = 0.0;
        sccurset(4,41);
        printf("0     ");
        }
surface_menu();
ShowCursor();
for ( ; quit != 1 ; ) {
        GetPosBut (&buttonStatus, &horiz, &vert);
        c = kbhit();
        if ((buttonStatus+c) != 0) {
                if ( c != 0 ) {
                        c = getch();
                        switch (c) {
                                case '1':
                                        c = 1;
                                        break;
                                case '2':
                                        c = 2;
                                        break;
                                case '3':
                                        c = 3;
                                        break;
                                case '4':
                                        c = 4;
                                        break;
                                default:
                                        c = 8;
                                }
                        }
                else {
                        if ( (horiz >= 437) && (horiz <= 624) ) {
                                if ( (vert >= 119) && (vert <= 272) )
                                        c = (vert - 119 ) / 42 + 1;
                                else c = 8;
                                }
                        else c = 8;
                        }
                switch (c) {
                        case 1:
                                wave_surf(climax,map);
                                flag = 1;
```

```
                        break;
                case 2:
                        gauss_surf(climax,map);
                        flag = 1;
                        break;
                case 3:
                        exp_ramp(climax,map);
                        flag = 1;
                        break;
                case 4:
                        quit = 1;
                        break;
                default:
                        ;
                }
        }
    }
/*clear window*/
HideCursor();
if (flag == 1) drawmap(*climax,map,1);
for(i=6;i<20;i++) {
        sccurset(i,54);
        printf("                          ");
        }
main_menu();
ShowCursor();
}

surface_menu()
{
int i,j,c,row,col;

/* Draw box */
scbox(6,54,19,78,15,0,LBLUE);
sccurset(7,55);
printf("   SURFACE SELECTION   ");
col=55;
c = 0xc4;
for(i=0;i<23;i++) {
        sccurset(8,col++);
        printf("%c",c);
        }
sccurset(9,55);
c = 0x31;
scattrib(LGREEN,BLACK,c,1);
sccurset(9,57);
```

```
printf("Wave surface          ");
sccurset(10,55);
printf("  z = A cos(wx)cos(wy) ");
col=55;
c = 0xc4;
for(i=0;i<23;i++) {
        sccurset(11,col++);
        printf("%c",c);
        }
sccurset(12,55);
c = 0x32;
scattrib(LGREEN,BLACK,c,1);
sccurset(12,57);
printf("Gaussian surface      ");
sccurset(13,55);
printf("  z = A exp(-h^2 r^2)  ");
col=55;
c = 0xc4;
for(i=0;i<23;i++) {
        sccurset(14,col++);
        printf("%c",c);
        }
sccurset(15,55);
c = 0x33;
scattrib(LGREEN,BLACK,c,1);
sccurset(15,57);
printf("Exponential ramp      ");
sccurset(16,55);
printf("  z = A exp(-h^2 y^2)  ");
col=55;
c = 0xc4;
for(i=0;i<23;i++) {
        sccurset(17,col++);
        printf("%c",c);
        }
sccurset(18,55);
c = 0x34;
scattrib(LGREEN,BLACK,c,1);
sccurset(18,57);
printf("End                   ");
}

wave_surf(climax,map)
float *climax;
float huge *map;
{
```

```
long row,col;
float a,tx,ty;
float wx,wy,cx,cy;

wave_para(&a,&tx,&ty);
if ( a > fabs(*climax) ) {
        *climax = a;
        sccurset(4,41);
        printf("       ");
        sccurset(4,41);
        printf("%5d",(int)*climax);
        }
wx = 2.*PI/tx;
wy = 2.*PI/ty;
for(row=0;row<128;row++) {
        cy = (float)cos((double)(127-row)*(double)wy);
        for(col=0;col<128;col++) {
                cx = (float)cos((double)col*(double)wx);
                map[col+row*128] += (a*cx*cy);
                }
        }
clr_par_win();
}

gauss_surf(climax,map)
float *climax;
float huge *map;
{
float a,r;
char type;
int ux,uy,lx,ly;
int xc,yc,row_start,row_end,col_start,col_end;
long row,col;
float h,d,z,sign;
PT startpt,endpt;

exp_para(2,&a,&r,&type,&xc,&yc);
if ((type == 'H') || (type  == 'h')) sign = 1.;
else
        sign = -1.;
if ( a > fabs(*climax) ) {
        *climax = a*sign;
        sccurset(4,41);
        printf("       ");
        sccurset(4,41);
        printf("%5d",(int)*climax);
```

```
        }
row_start = 127 - yc - (int)r;
if (row_start < 0) row_start = 0;
row_end = 127 - yc + (int)r + 1;
if (row_end > 127) row_end = 127;
col_start = xc - (int)r;
if (col_start < 0) col_start = 0;
col_end = xc + (int)r + 1;
if (col_end > 127) col_end = 127;
ux = col_start*3 + 29;
uy = row_start*2 + 73;
lx = col_end*3 + 29;
ly = row_end*2 + 73;
startpt.x = endpt.x = ux;
startpt.y = uy;
endpt.y = ly;
grline(&startpt,&endpt,7);
endpt.x = lx;
endpt.y = uy;
grline(&startpt,&endpt,7);
startpt.x = lx;
startpt.y = ly;
grline(&startpt,&endpt,7);
endpt.x = ux;
endpt.y = ly;
grline(&startpt,&endpt,7);
ux = (xc*3 + 29) / 8;
uy = ((127-yc)*2+73) / 14;
sccurset(uy,ux);
printf("2%c",type);
h = 1.730818383 / r;
h = h*h;
r = r*r;
yc = 127 - yc;
for(row=row_start;row<row_end;row++) {
        for(col=col_start;col<col_end;col++) {
                d = (row-yc)*(row-yc)+(col-xc)*(col-xc);
                if (d <= r)  {
                        z = a / (float)exp((double)(h*d));
                        map[col+row*128] += (sign*z);
                        }
                }
        }
clr_par_win();
}
```

```
exp_ramp(climax,map)
float *climax;
float huge *map;
{
float a,l;
char type;
int ux,uy,lx,ly;
int xc,yc,row_start,row_end,col_start,col_end;
long row,col;
float d,h,sign,*buf;
PT startpt,endpt;

exp_para(3,&a,&l,&type,&xc,&yc);

if ((type == 'H') || (type  == 'h')) sign = 1.;
else
        sign = -1.;
if ( a > fabs(*climax) ) {
        *climax = a*sign;
        sccurset(4,41);
        printf("       ");
        sccurset(4,41);
        printf("%5d",(int)*climax);
        }
buf = (float *)malloc(128*sizeof(float));
for(row=0;row<128;row++) buf[row] = 0.0;

row_start = 127 - yc - (int)l;
if (row_start < 0) row_start = 0;
row_end = 127 - yc + (int)l + 1;
if (row_end > 128) row_end = 128;

/*col_start = xc - (int)l;
if (col_start < 0) col_start = 0;
col_end = xc + (int)l + 1;
if (col_end > 127) col_end = 127;
ux = col_start*3 + 29;
lx = col_end*3 + 29;*/

startpt.x = 29;
endpt.x = 412;
startpt.y = endpt.y = row_start*2+73;
grline(&startpt,&endpt,7);
startpt.y = endpt.y = row_end*2+73;
grline(&startpt,&endpt,7);
```

```
ux = (xc*3 + 29) / 8;
uy = ((127-yc)*2+73) / 14;
sccurset(uy,ux);
printf("3%c",type);

h = 1.730818383 / l;
h = h*h;
for(row=row_start;row<row_end;row++) {
                d = (row-127+yc)*(row-127+yc);
                buf[row] = a / (float)exp((double)(h*d));
                buf[row] *= sign;
                }
for(row=row_start;row<row_end;row++)
        for(col=0;col<128;col++)
                        map[col+row*128] += buf[row];
clr_par_win();
}

clr_par_win()
{
int i;
for(i=7;i<18;i++) {
        sccurset(i,52);
        printf("                                ");
        }
surface_menu();
ShowCursor();
}


wave_para(arg1,arg2,arg3)
float *arg1,*arg2,*arg3;
{
int i;
char buf[80];
HideCursor();
/* Draw box */
scbox(7,52,17,79,15,0,LYELLOW);
/*clear box*/
for(i=8;i<17;i++) {
        sccurset(i,53);
        printf("                              ");
        }

sccurset(8,53);
printf("  z = A cos(wx)cos(wy) ");
```

```
sccurset(10,53);
printf("Magnitude A = ");
get_resp(10,67,buf);
*arg1 = (float)atof(buf);
sccurset(11,53);
printf("Wavelength (T=2");
i = 0xe3;
sccurset(11,68);
printf("%c",i);
sccurset(11,69);
printf("/w) along");
sccurset(12,56);
printf("x-axis = ");
get_resp(12,65,buf);
*arg2 = (float)atof(buf);
sccurset(13,56);
printf("y-axis = ");
get_resp(13,65,buf);
*arg3 = (float)atof(buf);
sccurset(16,53);
printf("Processing ...");
}

exp_para(type,arg1,arg2,arg3,arg4,arg5)
float *arg1,*arg2;
char *arg3;
int type,*arg4,*arg5;
{

int i,quit=0;
int c;
int horiz, vert;
int buttonStatus;
char buf[80];

HideCursor();
/* Draw box */
scbox(7,52,17,79,15,0,LYELLOW);
/*clear box*/
for(i=8;i<17;i++) {
        sccurset(i,53);
        printf("                            ");
        }
sccurset(8,53);
if (type == 2 )
```

```
        printf("  z = A exp(-h^2 r^2)  ");
else
        printf("  z = A exp(-h^2 y^2)  ");
sccurset(9,53);
printf("Magnitude A = ");
get_resp(9,67,buf);
*arg1 = (float)atof(buf);
sccurset(10,53);
printf("Cut-off length (5%% of A)");
sccurset(11,56);
printf("l = ");
get_resp(11,60,buf);
*arg2 = (float)atof(buf);
scbox(12,57,16,74,15,0,LYELLOW);
sccurset(13,58);
printf("Hill or Valley ?");
sccurset(15,62);
printf("(H / V)");
ShowCursor();
quit = 0;
for ( ; quit != 1 ; ) {
        GetPosBut (&buttonStatus, &horiz, &vert);
        c = kbhit();
        if ((buttonStatus+c) != 0) {
                if ( c != 0 ) {
                        c = getch();
                        switch (c) {
                                case 'H':
                                case 'h':
                                case 'V':
                                case 'v':
                                        *arg3 = c;
                                        quit = 1;
                                        break;
                                default:
                                        ;
                                }
                        }
                else {
                        if ( (horiz >= 496) && (horiz <= 551) ) {
                                if ( (vert >= 210) && (vert <= 224) ) {
                                        if ( horiz <= 524 ) *arg3 = 'h';
                                        else *arg3 = 'v';
                                        quit = 1;
                                        }
                                }
```

```
                        }
                }
        }
/*              sccurset(11,70);*/
HideCursor();
for (i=12;i<=16;i++) {
        sccurset(i,57);
        printf("                        ");
        }
while (buttonStatus != 0)
        GetPosBut (&buttonStatus, &horiz, &vert);
sccurset(12,53);
printf("Location of the center");
ShowCursor();
quit = 0;
for ( ; quit != 1 ; ) {
        GetPosBut (&buttonStatus, &horiz, &vert);
        if (buttonStatus != 0) {
                if ( (horiz >= 29) && (horiz <= 412) ) {
                        if ( (vert >= 73) && (vert <= 328) ) {
                                *arg4 = (horiz - 29) / 3;
                                *arg5 = 127 - ((vert - 73) / 2);
                                quit = 1;
                                }
                        }
                }
        }
HideCursor();
sccurset(15,56);
printf("x = %3d, y = %3d",*arg4,*arg5);
sccurset(16,53);
printf("Processing ...");
}

get_resp(row,col,buf)
char *buf;
int row,col;
{
int key=0,i,error=0;

for ( ; error == 0 ; ) {
        begin:
        for ( ; kbhit()==0 ; ) ;
        buf[0] = getch();
        if ( (buf[0] != 13) && (buf[0] != 8) ) {
                sccurset(row,col++);
```

```
                printf("%c",buf[0]);
                i = 1;
                for ( ; key == 0 ; ) {
                        for ( ; kbhit()==0 ; ) ;
                        buf[i] = getch();
                        if (buf[i] == 8) {
                                buf[i] = 32;
                                sccurset(row,--col);
                                printf("%c",buf[i--]);
                                if (i==0) goto begin;
                                }
                        else {
                                sccurset(row,col++);
                                printf("%c",buf[i]);
                                if (buf[i++] == 13) {
                                        buf[--i] = 0;
                                        key=1;
                                        }
                                }
                        }
                error = 1;
                }
        }
}
drawmap(climax,map,clearflag)
float climax;
float huge *map;
int clearflag;
{
long row,col;
float z;
int *lookup,i,index;
HideCursor();
lookup = (int *)malloc(201*sizeof(int));
for(i=0;i<10;i++) lookup[i]=8; /*generate lookup table*/
for(i=10;i<30;i++) lookup[i]=5;
for(i=30;i<50;i++) lookup[i]=4;
for(i=50;i<70;i++) lookup[i]=3;
for(i=70;i<85;i++) lookup[i]=2;
for(i=85;i<91;i++) lookup[i]=1;
for(i=91;i<109;i++) lookup[i]=0;
for(i=110;i<115;i++) lookup[i]=9;
for(i=115;i<130;i++) lookup[i]=10;
for(i=130;i<150;i++) lookup[i]=11;
```

```
for(i=150;i<170;i++) lookup[i]=12;
for(i=170;i<190;i++) lookup[i]=13;
for(i=190;i<=200;i++) lookup[i]=14;

climax = (float)fabs((double)climax);
if (clearflag) clearmap();
for (row=0;row<128;row++) {
        for (col=0;col<128;col++) {
                z = map[col+row*128];
                if ( z == OCCLUDED)
                drawpoint(row,col,6);
                else if ( z == UNSCANNED)
                        drawpoint(row,col,7);
                else {
                        z = z / climax;
                        z = (z+1.)*100.;
                        index = z;
                        if ( lookup[index] != 0 ) drawpoint(row,col,lookup[index]);
                        }
                }
        }
ShowCursor();
}

clearmap()
{
int i;
for(i=5;i<24;i++) {
        sccurset(i,3);
        printf("                                                                ");
        }
drawbox();
}

drawpoint(row,col,color)
long row,col;
int color;
{
PT startpt,endpt;
startpt.x = col*3 + 29;
startpt.y = endpt.y = row*2 + 73;
endpt.x = startpt.x + 2;
grline(&startpt,&endpt,color);
startpt.y = endpt.y += 1;
grline(&startpt,&endpt,color);
}
```

```
/*
edge.c --
        noise insensitive edge detector for range data
*/

#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <string.h>
#include "utility.h"
#include "matrix.h"

#define   RAD       .01745329252

/*SCANNER SPECIFICATIONS*/
float   INTV,VERTSC,HORZSC,TILT,XSTART,DEG_X,DEG_Y,JUMP_THRESH;
unsigned NUM_ROW,NUM_COL;
int     get_planes(),eigsmall();

main()
{
char            in_file[15];
unsigned        win_size;
float           noise,noise_thresh,edge_thresh;

printf("input filename = ");
scanf("%s",in_file);
printf("standard deviation of noise = ");
scanf("%f",&noise);
printf("noise threshold = ");
scanf("%f",&noise_thresh);
printf("crease edge threshold = ");
scanf("%f",&edge_thresh);
printf("jump edge threshold = ");
scanf("%f",&JUMP_THRESH);
printf("window size = ");
scanf("%u",&win_size);
edge(in_file,win_size,noise,noise_thresh,edge_thresh);
}
```

```
edge(in_file,win_size,noise,noise_thresh,edge_thresh)
char            *in_file;
unsigned        win_size;
float           noise,noise_thresh,edge_thresh;
{
int             open(),fp1;
FILE            *fopen(),*fp;
char            char_buf[15];
int             i,j,tmp_int,iloop_end,jloop_end,edge_dir;
float           *sn,edge;
unsigned char *in_mat,*out_mat;

fp=fopen("edge.dat","r");
fscanf(fp,"%u %u",&NUM_ROW,&NUM_COL);
fscanf(fp,"%f %f",&VERTSC,&HORZSC);
fscanf(fp,"%f",&INTV);
fscanf(fp,"%f",&TILT);
fclose(fp);
DEG_X=HORZSC/NUM_ROW;
DEG_Y=VERTSC/NUM_COL;
XSTART=-(HORZSC/2);
edge_thresh *= RAD;
edge_thresh = cos(edge_thresh);
if ( (NUM_ROW*NUM_COL) > 128*128 ) {
        printf("Warning: input file too large to handle\n");
        exit(1);
        }
sn=(float *)malloc(sizeof(float)*8*4);
in_mat=(unsigned char *)malloc(NUM_ROW*NUM_COL);
out_mat=(unsigned char *)malloc(NUM_COL);
if (out_mat == NULL) {
        printf("can't allocate enough memory\n");
        exit(1);
        }
for(i=0;i<NUM_COL;i++) out_mat[i]=0.0;
fp1=open(in_file,O_RDONLY);
setmode(fp1,O_BINARY);
if ( (read(fp1,in_mat,NUM_ROW*NUM_COL)) == -1) {
        printf("can't read from the input file\n");
        exit(1);
        }
close(fp1);
tmp_int=strcspn(in_file,".");
strncpy(char_buf,in_file,tmp_int);
char_buf[tmp_int]=0;
strcat(char_buf,".edg");
```

```
fp1=creat(char_buf,S_IREAD|S_IWRITE);
setmode(fp1,O_BINARY);
for(i=0;i<win_size/2;i++) write(fp1,out_mat,NUM_COL);
iloop_end = NUM_ROW-(win_size/2);
jloop_end = NUM_COL-(win_size/2);
for (i=win_size/2;i<iloop_end;i++) {
        for (j=win_size/2;j<jloop_end;j++) {
/* sample the surface normals of the adjacent regions centered at (i,j) */
                tmp_int =
                get_planes(sn,in_mat,i,j,win_size,noise,noise_thresh);
/* examine if the sampled regions belong to a same surface */
                if (tmp_int <= 1) out_mat[j] = 255.;
                else {
                        edge = 1.;
                        chk_edge(sn,tmp_int,&edge,&edge_dir,edge_thresh);
                        if (edge > edge_thresh ) out_mat[j]=0;
                        else {
                                if (edge < -.1) out_mat[j]=200;
                                else {
                                        out_mat[j]=edge_dir;
                                        }
                                }
                        }
                }
        write(fp1,out_mat,NUM_COL);
        printf("i=%d\n",i);
        }
for(i=0;i<NUM_COL;i++) out_mat[i]=0.0;
for(i=0;i<win_size/2;i++) write(fp1,out_mat,NUM_COL);
close(fp1);
free(in_mat);
free(out_mat);
free(sn);
}
```

```
chk_edge(sn,n,edge,edge_dir,edge_thresh)
float           *sn,*edge,edge_thresh;
int             n,*edge_dir;
{
int             i;
double          dou_temp;
float           *vector2,tmp_float,out[3];
n=n-1;
for (i=1;i<=n;i++) {
        vector2=sn+4*i;
        tmp_float = vdotv(sn,vector2,3);
        tmp_float = fabs(tmp_float);
        if ( fabs(fabs(sn[3])-fabs(vector2[3])) >= JUMP_THRESH)
                *edge=-1;
        else {
                if (tmp_float <= edge_thresh) {
                        if (tmp_float < *edge) {
                                *edge=tmp_float;
                                vcrossv(sn,vector2,out);
                                if (fabs(out[0]) <= 1.e-10) *edge_dir = 2;
                                else {
                                        dou_temp = out[2]/out[0];
                                        dou_temp = atan(dou_temp);
                                        if ((dou_temp>=-1.309)&&(dou_temp<-0.0436) )
                                                *edge_dir = 4;
                                        else if
                                        ((dou_temp>=-0.0436)&&(dou_temp<0.0436) )
                                                *edge_dir = 1;
                                        else if ((dou_temp>=.0436)&&(dou_temp<1.309))
                                                *edge_dir = 3;
                                        else
                                                *edge_dir = 2;
                                        }
                                }
                        }
                }
        }
if ((n>1) && (*edge != -1) ) chk_edge(sn+4,n,edge,edge_dir,edge_thresh);
}
```

```
int get_planes(sn,mat,cur_i,cur_j,win_size,alpha,threshold)
float           *sn,alpha,threshold;
unsigned char *mat;
int             cur_i,cur_j,win_size;
{

int             i,j,n,inc;
float           lambda,k,d_mean;
i=1; n=0; inc=2;
do {
        get_plane_eqa(sn,n,&lambda,i,&k,mat,cur_i,cur_j,&d_mean,win_size);
        lambda = lambda/(k*k*d_mean*d_mean*alpha*alpha);
        if (lambda > threshold) {
                if (inc != 1) {
                        j = i+1;
                get_plane_eqa(sn,n,&lambda,j,&k,mat,cur_i,cur_j,&d_mean,win_size);
                        lambda = lambda/(k*k*d_mean*d_mean*alpha*alpha);
                        if (lambda > threshold) {
                                if ( (j == 4) && (n == 0)) inc = 1;
                                }
                        else {
                                n++; }
                        }
                }
        else
                n++;
        i = i+inc;
        } while (i <= 7 );
return(n);
}


get_plane_eqa(sn,n,lambda,c,k,in_mat,cur_i,cur_j,d_mean,win_size)
float           *sn,*lambda,*k,*d_mean;
unsigned char *in_mat;
int             n,c,cur_i,cur_j,win_size;
{

int             i,j,index,target_i,target_j,sample_size,tmp_int;
float           dist,theta,phi,x_prime,y_prime,z_prime,length;
float           *x,*y,*z,mean_point[3],a[3][3],s[3][3];

for(i=0;i<3;i++)
        for(j=0;j<3;j++) a[i][j]=0.;
*d_mean=0.;
mean_point[0]=mean_point[1]=mean_point[2]=0;
```

```
index = 0;
sample_size = win_size/2-1;
tmp_int=0;
for(i=1;i<=sample_size;i++) tmp_int +=i;
x=(float *)malloc((unsigned)sizeof(float)*tmp_int);
y=(float *)malloc((unsigned)sizeof(float)*tmp_int);
z=(float *)malloc((unsigned)sizeof(float)*tmp_int);
for(i=0;i<sample_size;i++) {
        switch (c) {
                case 1:
                        target_i = (cur_i+i+2);
                        break;
                case 2:
                        target_i = (cur_i-i+sample_size);
                        break;
                case 3:
                        target_i = (cur_i+i-sample_size);
                        break;
                case 4:
                        target_i = (cur_i-i-2);
                        break;
                case 5:
                        target_i = (cur_i-i-2);
                        break;
                case 6:
                        target_i = (cur_i+i-sample_size);
                        break;
                case 7:
                        target_i = (cur_i-i+sample_size);
                        break;
                case 8:
                        target_i = (cur_i+i+2);
                }
        for(j=i; j>=0; j--) {
                switch (c) {
                        case 1:
                                target_j = (cur_j-j-1);
                                break;
                        case 2:
                                target_j = (cur_j+j-sample_size-1);
                                break;
                        case 3:
                                target_j = (cur_j+j-sample_size-1);
                                break;
                        case 4:
                                target_j = (cur_j-j-1);
```

```
                        break;
                case 5:
                        target_j = (cur_j+j+1);
                        break;
                case 6:
                        target_j = (cur_j-j+sample_size+1);
                        break;
                case 7:
                        target_j = (cur_j-j+sample_size+1);
                        break;
                case 8:
                        target_j = (cur_j+j+1);
                        }
                tmp_int = in_mat[NUM_COL*target_i+target_j];
                dist=(INTV/256.)*(float)tmp_int;
                *d_mean += dist;
                theta=RAD*(XSTART+((float)target_j*DEG_X));
                phi=RAD*(TILT+(float)target_i*DEG_Y);
                x[index]=dist*(sin(theta));
                y[index]=dist*(cos(theta))*(cos(phi));
                z[index]=-(dist*(cos(theta))*(sin(phi)));
                mean_point[0] += x[index];
                mean_point[1] += y[index];
                mean_point[2] += z[index++];
                }
        }
*d_mean /= index;
for (i=0;i<3;i++) mean_point[i] /= index;
for (i=0;i<index;i++) {
        x_prime = x[i] - mean_point[0];
        y_prime = y[i] - mean_point[1];
        z_prime = z[i] - mean_point[2];
        a[0][0] += (x_prime * x_prime);
        a[0][1] += (x_prime * y_prime);
        a[0][2] += (x_prime * z_prime);
        a[1][1] += (y_prime * y_prime);
        a[1][2] += (y_prime * z_prime);
        a[2][2] += (z_prime * z_prime);
        }
a[1][0]=a[0][1];
a[2][0]=a[0][2];
a[2][1]=a[1][2];
i=3;
tmp_int=3;
mateig(i,tmp_int,&(a[0][0]),&(s[0][0]));
i=eigsmall(&(a[0][0]),tmp_int);
```

```
*lambda = a[i][i];
for (j=0;j<3;j++) {
        sn[n*4+j] = s[j][i];
        }
sn[n*4+3]=*d_mean;
length=vector_length_3(mean_point);
*k = -sn[n*4+3]/length;
free(x);free(y);free(z);
}

int eigsmall(a,n)
float   *a;
int     n;
{
int     k,j;
float   p;
k=0;
p=*a;
for (j=1;j<n;j++)
        if (a[n*j+j] < p) {
                k = j;
                p=a[n*j+j];
                }
return(k);
}
```

```
/* mateig.c --
        to calculate eigen values and corresponding eigen vectors of a real symmetric
        matrix by the Jacobi method
*/

mateig(lra,n,a,s)
int             lra,n;
float   a[],s[];
{
double          fabs(),sqrt();
int             i,ia,ii,iia,iib,ij,ik,il,in,ip,iq,is,j,ja,ka,la,m;
float   anorm,app,aqq,b,c,constf,csn;
float   csn2,e,fnorm,snn,snn2,stc,u,v,w;

e = 0.000001;
constf = n;
in = 0;
for(j = 0; j <= n-1; j++)
        for(i = 0; i <= n-1; i++) {
                is = j * lra +i;  /* move columnwise to new element */
                if ((i-j) == 0) s[is] = 1.;  /* look for diagonal */
                else s[is] = 0.0;           /* set non-diagonal to Z */
                }  /* end of i loop */
          /* end of j loop */
        anorm = 0.0;
        for(j = 0; j <= n-1; j++)
                for(i = 0; i <= n-1; i++) {
                        ia = i * lra +j;              /* column */
                        if ((j-i) != 0)
                        anorm = anorm + a[ia] * a[ia];
                        } /* end of i loop */
         /* end of j loop */
        anorm = sqrt(anorm);
        fnorm = anorm * e;
        190:;
        anorm = anorm/constf;
        180:;
        for(iq = 1; iq <= n-1; iq++)
        {
        m = iq - 1;
        for(ip = 0; ip <= m; ip++) {
                ia = lra * iq + ip;                     /* row iq , col ip */
        ja = lra * ip + iq;             /* row ip , col iq */
                ka = lra * ip + ip;                     /* row ip , col ip */
        la = lra * iq + iq;             /* row iq , col iq */
        if ((fabs(a[ja]) - anorm) <  0.0) goto l21;
```

```
        in = 1;
        u = -a[ja];
                v = (a[ka] - a[la]) / 2.0;
                w = u/(sqrt(u*u + v*v));
                if (v <  0.0) w = -w;
        snn = w/(sqrt(2.0 * (1.0 + sqrt(1.0 - w*w))));
                snn2 = snn * snn;
                csn = sqrt(1.0 - snn2);
                for(i = 0; i <= n-1; i++) {
                        iia = lra * i + ip;     /* row i, col ip */
                   iib = lra * i + iq;          /* row i, col iq */
                        if ((i - ip) == 0) goto 1301;
                if ((i - iq)  == 0) goto 1301;
                b = a[iia] * csn - a[iib] * snn;
                a[iib] = a[iia] * snn + a[iib] * csn;
                a[iia] = b;
                1301:
                        c = s[iia] * csn - s[iib] * snn;
                        s[iib] = s[iia] * snn + s[iib] * csn;
                        s[iia] = c;
                }   /* end of i loop */
                csn2 = csn * csn;
        stc = snn * csn;
                app = a[ka]*csn2 + a[la]*snn2 - 2.0*a[ja]*stc;
                aqq = a[ka]*snn2 + a[la] * csn2 + 2.0* a[ja] *stc;
                a[ja] = (a[ka] - a[la])*stc + a[ja]*(csn2-snn2);
                a[ia] = a[ja];
                a[ka] = app;
                a[la] = aqq;
                for(i = 0; i <= n-1; i++) {
                ii = lra * ip + i;              /* row ip, col i */
                ij = lra * i + ip;              /* row i, col ip */
                ik = lra * iq + i;      /* row iq, col i */
                il = lra * i + iq;      /* row i, col iq */
                a[ii] = a[ij];
                a[ik] = a[il];
                        }     /* end of i loop */
                121: ;
        }  /* end of ip loop */
        }                 /* end of iq loop */
if ((in-1) != 0) goto 1600;
in = 0;
goto 180;
1600: if((anorm-fnorm) > 0.0) goto 190;
}
```