RANGE IMAGE INTEGRATION AND HIERARCHICAL DISTANCE MAPS FOR SENSOR-BASED COLLISION DETECTION AND PATH PLANNING

by

Derek Jung

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE
in the School
of
Engineering Science

SIMON FRASER UNIVERSITY
August 1997

0-612-24168-8

APPROVAL

Name:	Derek Jung
Degree:	Master of Applied Science
Title of thesis :	Range Image Integration and Hierarchical Distance Maps for Sensor-Based Collision Detection and Path Planning

Examining Committee: Dr. Andrew Rawicz
Professor, Engineering Science, SFU, Chairman

Dr. Kamal Gupta
Associate Professor, Engineering Science, SFU
Senior Supervisor

Dr. Shahram Payandeh
Associate Professor, Engineering Science, SFU
Internal Supervisor

Dr. Ze-Nian Li
Associate Professor, Computing Science, SFU
Examiner

Date Approved: 2nd Sept. '97

Abstract

We investigate two topics toward efficient sensor-based collision detection and path planning in real-world environments. The first topic deals with the gathering of real-world data via integration of multiple-view laser range images, and the second topic deals with the creation and use of hierarchical distance maps.

The integration of range images from multiple views generates three-dimensional spatial occupancy (voxel) models. As opposed to CAD or geometric models, spatial occupancy models are closer to raw data, thereby they can be easily derived from raw range images. Furthermore, they can easily and accurately represent complex real-world environments. Two integration methods are examined: direct mapping, which accurately represents the surface shapes of objects; and peeling, which creates a model that represents the space occupied by objects, in addition to unscanned areas. The resulting voxel array may then be used for collision detection. However, for efficiency and speed, these voxel maps can be pre-processed into distance maps – each free pixel is assigned a value equal to the closest object, at the expense of significant memory requirements. We propose and implement a novel hierarchical distance map for collision detection. It is based on the standard octree representation and is called the octree distance map (ODM). The ODM represents distance information in a hierarchical manner, yielding efficient memory use while maintaining low cost in execution time. Two algorithms are presented, one for ODM creation and the other for ODM-based collision detection.

Experiments with both range image integration and ODM's were promising; ODM's in particular proved to be an excellent compromise between using array-based distance maps (high speed, high memory requirements) and regular octrees (low speed, low memory requirements) for collision detection.

Acknowledgements

This research has been supported by a grant from the Institute for Robotics and Intelligent Systems (IRIS), a Federal Centre of Excellence in Canada.

I wish to thank Dr. Kamal Gupta, Simon Fraser University, for his time, support, and patience. In addition, thanks to Dr. Shahram Payandeh of Simon Fraser University, for accepting a position on my Thesis Committee, and to Dr. Ze-Nian Li of SFU Computing Science for accepting the position of Examiner. I must also thank Bill Woods for all his time and effort in helping me with hardware and material matters, Chao Cheng for keeping the computers up and running, and the people at Technical Arts, Inc., for answering my range scanner questions. I also wish to thank Dr. Joe Peters and Robert Hwang of the School of Computing Science at Simon Fraser University for their comments and suggestions.

Finally, thanks to all my family, friends, and fiancée for their support (and for not letting me give up on this!).

Contents

List of Tables

List of Figures

Chapter 1

Overview

The use of detailed, real-world three-dimensional data for collision avoidance and path planning in robotics is very desirable and thus a significant area of research. While using CAD-type data to model real-world environments is currently an acceptable approach, the growing number of applications for robotics expands the range of possibilities for real-world situations such that modeling these situations using CAD techniques can be very difficult. In addition, the power of computing hardware today removes many limitations on the computational requirements for compiling sensed data into a usable model.

Using real-world data, this thesis addresses these two issues of robot path planning: (1) generation of a usable, efficent model of the workspace and (2) algorithms for performing collision detection with this model. In our view, a robot workspace may contain any number of objects of all shapes and sizes. While specific situations may limit the number and class of objects to provide clean, controlled environments and thus simplify model generation and collision detection, we decided to investigate the strategies that would provide a system usable in the widest range of environments possible. For instance, the workspace model generation and collision detection algorithms both use voxelized space and octrees to model the environment. Thus, any object or set of objects can be modeled with the only limiting factor being the chosen resolution.

This section provides a brief introduction to these two issues, workspace model generation and collision detection using the workspace model; both issues are discussed in far greater detail later in the thesis. They are separately presented in this thesis because, although both are important issues in robot path planning, the two issues are complementary. The common link between the two is that a generated workspace model would be passed to the collision detection algorithm in an actualized system; this was, unfortunately, not achieved due to experimental delays in the project. However, path planning algorithms using CAD environments have been developed at SFU (Gupta and Zhu 1995) and can be adapted to use voxel maps such as those generated using algorithms presented here. The thesis provides concrete evidence that, given more sophisticated hardware and further development, a fully integrated system using the concepts and algorithms described here is achievable.

1.1 Generation of Workspace Models via Range Image Integration

The first part of this thesis discusses the investigation of the use of laser range images in the generation of a workspace model. Specifically, multiple range images are taken from sufficiently different points of view and integrated via geometric calculations into a discrete "voxel map" or 3-d bitmap. The resulting voxel map, once all images are integrated, is the workspace model. Note that the model is a volumetric model, not a CAD model, for reasons given above.

Chapter 2 provides a more detailed introduction on this subject, while Chapter 3 discusses previous work in this field and introduces range scanning. In Chapter 4, the concepts behind range image integration, and in particular the integration system used for this thesis, are presented as well as the algorithms investigated. Chapter 5 discusses the experiments performed and results obtained, and Chapter 6 presents the conclusions for this part.

1.2 Octree-Based Distance Map Representation for Path Planning

Part 2 of the thesis investigates the issue of using volumetric workspace models for collision detection and path planning. Previous efforts in the use of distance maps for path planning (Latombe 1991) have shown promise, particularly in terms of performance, but have been less than ideal in terms of efficiency in memory use. In this thesis, an octree-based representation, the *Octree Distance Map*, is presented and shown to be (1) usable for collision detection and (2) efficient in memory usage and collision detection performance. This part of the thesis is based primarily on an upcoming article in the *Journal of Robotic Systems* (Jung and Gupta 1997).

Chapter 7 introduces this topic in greater detail. Chapter 8 provides background information about previous related work and about octree representations. Chapter 9 presents the *Octree Distance Map* and algorithms for its generation and use in collision detection. Chapter 10 discusses the experiments performed using the Octree Distance Map concept, and Chapter 11 presents the conclusions drawn from the experiments.

1.3 Contributions of Thesis

This thesis has contributed in terms of both function and theory. The theoretical contributions include the direct-mapping and peeling algorithms for range image integration, the concept of the octree distance map (ODM), and algorithms for creating ODM's and using ODM's in sensor-based collision detection and path planning. Functionally, the thesis has produced an experimental system for the acquisition and integration of multiple-view range images. In addition, the above algorithms have been implemented and experiments have been performed using these algorithms.

Part I

Spatial Occupancy Recovery by the Integration of Range Images

Chapter 2

Introduction

Most robot path-planning programs in the past have operated on CAD models of robot workcells, or on completely artificial environments (Latombe 1991). CAD models are simple representations, but obtaining a CAD model of a real environment based on sensed data is quite difficult, simply because the modeling of complicated real-world objects using CAD primitives is a non-trivial problem. This problem, in fact, is the basis of a great body of research. An alternative to using CAD models is to build a *spatial occupancy* model, that is, a model which maps the volumes or spaces occupied by objects in the workcell. Such models are easier to obtain from sensed data than CAD models. An example of a volumetric model is a binary *voxel* array. A voxel is a elementary volume (3-d) element, analogous to a pixel in 2-d. By setting voxels in obstacle or object space to ON and voxels in free space to OFF, the spatial occupancy model of a robot workcell is obtained.

Acquiring information about a robot workspace can be accomplished in a number of ways, using many different types of sensors: sonar, video cameras, force sensors, and so on. One such type of sensor is the laser range scanner, which has the added advantages of direct three-dimensional information capture and high accuracy. However, a single range image in most cases will not adequately describe a workspace, and therefore we must use multiple scans and find a way to merge or fuse the resulting images into a single representation.

The process of merging 3-D data contained in multiple range images into one 3-D workspace model lies in the realm of computer vision research. However, most 3-D vision research has concentrated on object recognition aspects rather than workspace or environment modeling. In object recognition, one is generally concerned with a single object and matching it among models with geometric descriptions. For the workspace modeling problem, however, the main concern is the actual spatial occupancy of each object – and there may be multiple objects – in the workspace; the geometric properties of the objects are of interest only insofar as the effect of such properties on spatial occupancy. In addition, we have a contrast in the ideologies behind the nature of the acquired models: object recognition deals with a model of a single object, whereas I am interested in a model of an entire environment – both occupied and unoccupied spaces. Research in mobile robot navigation has dealt with mapping the environment, but with different types of sensor systems, e.g., sonar/intensity images, and in different situations, e.g., on a different spatial scale.

In this first part of the thesis, I provide an overview of previous related research in image integration. I briefly present the concepts behind range imaging, and then discuss the calibration processes required in our particular system for us to obtain the proper transforms for range image integration. Next, I examine the actual problem of range image integration. I propose an algorithm, called *peeling*, which fuses range images from multiple views given the geometric transforms for those views. The result of the algorithm is a spatial occupancy model (in the form of a *voxel array*, a voxel being a single volume element) of the robot workcell; this model can then be input to a path planner. I also take a look at another method of range image integration, *direct mapping*, which combines the images into a voxel array but provides only a surface modelling of the objects in the workcell, not a spatial occupancy model. I then describe the implementation of the prototype system and present results of the experiments.

Chapter 3

Background

3.1 Range Scanning

Many methods for obtaining 3-d data exist. For instance, one can use sonar, which involves emitting sound waves and timing their return to determine the distance of the reflecting surface. Stereo systems are common, and exist in two varieties. The first uses two or more intensity cameras separated by a known distance. All of the cameras capture an image of the same object or scene at the same time and, based on the disparities between matched features in the multiple images and the geometry of the cameras, the distance to points in the images may be determined. The second variety of stereo vision is called *photometric stereo.* Photometric stereo involves a single intensity camera, which captures images of an object or scene from different points of view (or images of a moving object). 3-d data is again calculated using disparity of matched features and geometry. Shape data can also be approximated from single intensity images through shape-from-shading – inferring 3-d data from light intensity gradients.

The method used in this thesis, laser range scanning, produces 3-d data in the form of *range images.* A range image is a 2-d image whose pixels have values relative to the range (or distance) between the scanned surface and some reference point. Figure 3.1 provides an example of the information contained in a range image.

Laser range scanning has a number of advantages over the above methods: First, no feature matching between images is required, as long as the geometric transformations between each scanner position are known. Second, only one camera is required.

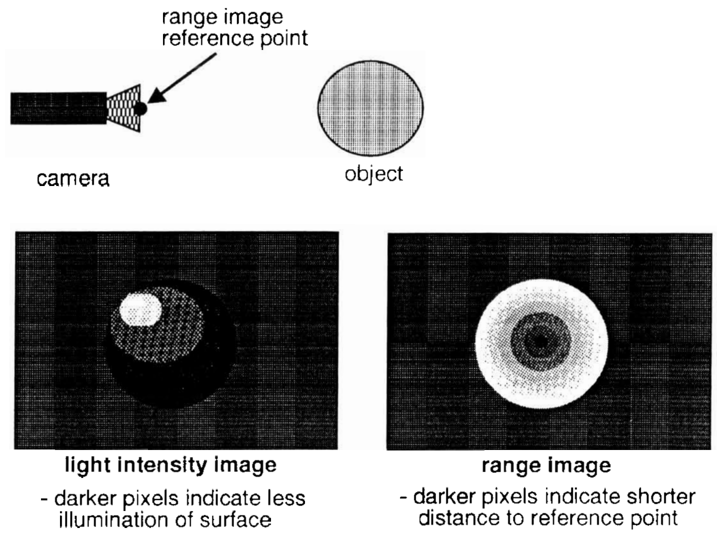

range image reference point
camera
object
light intensity image
- darker pixels indicate less illumination of surface
range image
- darker pixels indicate shorter distance to reference point

Figure 3.1: Example of a range image vs. an intensity image.

Third, no special lighting conditions are required, as long as the camera may discern reflected laser light from ambient lighting. Fourth, colour differences on the surface of an object do not affect accuracy (with certain exceptions: black, for example, tends not to reflect light very well and thus our laser scanner has problems with black surfaces.). Finally, and most significantly, the range image obtained from the laser range scanner contains explicit and highly accurate 3-d data – no further manipulation of the image is required.

Two types of range scanners are most common: *time-of-flight* scanners and light-stripe scanners. Time-of-flight scanners use a principle similar to sonar: a pulse of laser light is emitted and the time required for it to hit and be reflected by a surface is measured. These scanners are extremely accurate, but also extremely expensive. Light-stripe or *White* scanners are slightly less accurate and thus less expensive. Our White scanner has two components: a laser source and a CCD camera. The laser source has two mirrors: a *spreader* mirror, and a *director* mirror (see Figure 3.2) The spreader mirror oscillates quickly, such that when a laser beam contacts it, the mirror reflects the beam at various angles, thus creating the light plane. This plane is then positioned by the director mirror to fall onto the surfaces to be imaged. When the light plane hits a surface, the result is a light stripe on the surface, which is imaged by the CCD camera (see Figure 3.3). If there are variations in the height of the surface, these variations will show up on the image as a displacement of the light stripe. By previously measuring parameters such as the angles and heights of the laser and camera from the surface (thus defining the geometrical situation), the heights of the variations may be calculated from the image. An area scan of an entire scene can be performed by having the director mirror gradually step the laser stripe across the complete scene. The image of the stripe is passed from the CCD camera to a frame grabber, which digitizes the image. The digitized values are then fed into a computer, which calculates the range (distance) to each scanned point. Accuracy can be improved by taking repeated scans for a given position and averaging the range values associated with each point.

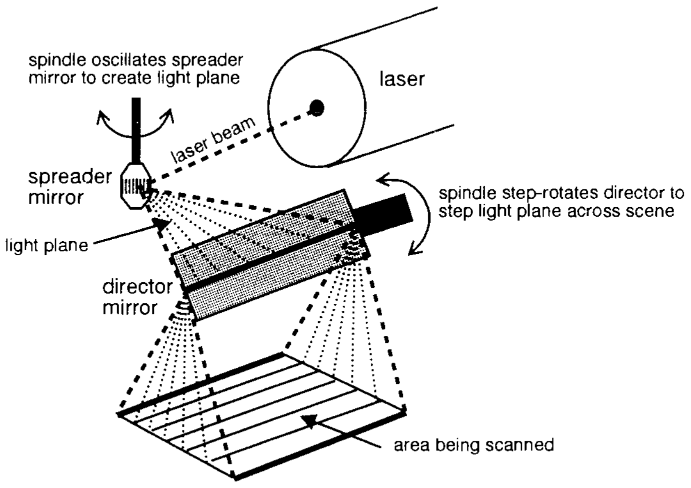

spindle oscillates spreader mirror to create light plane
laser
laser beam
spreader mirror
spindle step-rotates director to step light plane across scene
light plane
director mirror
area being scanned

Figure 3.2: The spreader and director mirrors on the White scanner.

3.2 Previous Work

A great deal of active research exists in the area of image integration. While my own concentration has been the integration of range images only, many researchers have investigated fusing range and intensity data in the same system, and the integration of other types of data, such as sonar, from which 3-d structure may be obtained. For instance, Asada (Asada 1990) combines both intensity and range data to build

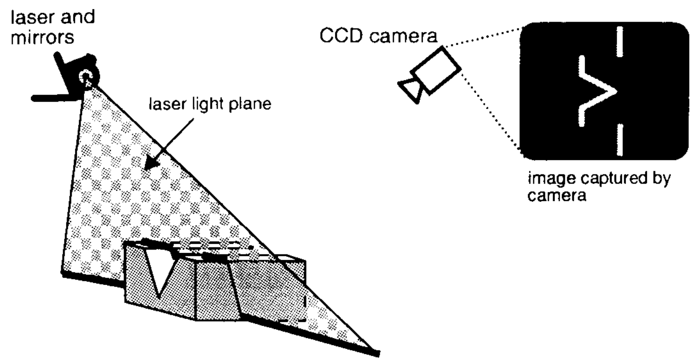


Figure 3.3: Operation of the White scanner.

environment maps for a mobile robot. The method in this paper uses a single range image to produce a "height map" which represents the environment in 3-D. This height map is then segmented and used with intensity data to identify and classify the obstacles located in the image as artifical objects (objects with planar surfaces), natural objects, or uncertain. Unfortunately, the approach is limited to mapping convex, floor-mounted objects. Grandjean and Robert de Saint Vincent (Grandjean et al. 1989) have proposed a method for fusing both range data from a laser range scanner and photometric data from a stereo (intensity-image) system. Its results, scene descriptions consisting of a set of planar faces, is more suited for polyhedral and geometric modeling. The method uses "extended Kalman filtering" to fuse lower-level primitives (points/pixels) into higher ones (3-D lines, planes) and for calibrating the transformations between reference frames. Leonard and Moran (Leonard and Moran 1992) describe a geometric approach for the integration of sonar data in order to reconstruct underwater 3-D scenes. They use an approximation of the geometrical characteristics of high-frequency acoustic scattering to try to recover explicit geometric surface descriptions of objects.

In range image integration research, one can easily identify two main areas of concentration: (i) the production of a geometrical or CAD model, and (ii) the production of a spatial occupancy model in the form of a voxel array or octree. (Octrees are discussed in detail in the second part of this thesis.) The largest body of work in image integration, particularly in range image integration, deals with the extraction of geometrical models from fused images. These models are generally then used for object recognition and pattern matching, or to build CAD models of objects. The methods proposed usually involve some form of segmentation or feature extraction. For example, Succi and others (Succi et al. 1990) describe a system for extracting features from a sequence of range data. The system performs volumetric integration on the range data, and then detects planes and vertices from a "Polynet" 3-D superficial representation built from the volumetric one. Stenstrom and Connolly (Stenstrom and Connolly 1986) generate polyhedral wire frames from multiple range images by extracting line segments from each view, and then transforming them into a global frame. Herman (Herman 1985) produces descriptions of polyhedral objects from range data, in the form of 3-D primitives (vertices, lines, planes, etc.) as well as topological and geometric relationships. Yao and others (Yao, Podhorodeski, and Zuomin 1993) present a multiple-view range image integration method based on partial geometric modeling for each range image. A global model is updated after each partial model is generated, with the final result being a complete 3-D geometric description of objects in the scene, once all the range images have been considered. There are many other such examples, as well as examples of raw range data processing for object recognition (e.g., Lin and Wee (Lin and Wee 1985) apply a generalized Hough transform on range data in order to recognize or locate 3-D objects).

Stenstrom and Connolly (Stenstrom and Connolly 1992) demonstrate a method for producing solid 3-D models from multiple range or intensity images, or from digitized line drawings. For each image, the algorithm first finds all pixels which form part of an edge and groups these into chains. Next, *1-cycles*, or sets of edges where each endpoint is common to only two edges in the set, are located from the edges identified in the image. Each 1-cycle defines a finite area in a plane. 1-cycles are then extruded orthogonally in the view direction from the image plane's physical position to produce *cycle volumes* or *2-cycles*. (2-cycles are defined as a set of faces where edges of faces are incident on only two faces in the set, and cycle volumes are 2-cycles formed by the extrusion of 1-cycles.) Cycle volumes from multiple views are then intersected to obtain a bounding volume for the object in question. The end result is a closed 2-cycle object model which fully bounds the object and approximates the object by a planar solid.

Previous work at Simon Fraser University involving the integration of range images is documented in (Xu 1992). Xu generates a 3-D boundary representation or *b-rep* description of a polyhedral object by fusing multiple synthetic range images of the object, taken from multiple viewpoints. In his approach, the rigid-body transformations between each view are first determined through a matching algorithm which identifies and relates *triple branch structures* (structures containing an object's partial geometric information, formed by three intersecting and noncoplanar edges) in the range images. Once the relationships between the views are known and the features (vertices, edges, and faces) in each view have been identified, the features are transferred to a global reference frame based on the rigid-body transformations. Duplicate features are checked for and removed. The resulting geometrical *b-rep* model is a list of vertices (and their $x - y - z$ coordinates), a list of edges (with start and end vertices and bordering faces), and a list of faces (with a list of vertices for each face). These three lists provide a complete description of a polyhedral object.

The second main branch of range image integration research, the generation of volumetric or spatial occupancy models (such as voxel maps or octrees) from multiple range images, has also attracted attention from researchers. Some methods used for range image integration have been derived from earlier efforts using intensity images. An early example is (Martin and Aggarwal 1983), which discusses a method for obtaining models of three-dimensional objects in multiple intensity images. In each image, the *occluding contour* of the object is determined; the occluding contour is the boundary in the image plane of the silhouette of the object, with the silhouette generated by intensity-thresholding the image. Another way to conceptually define the occluding contour is to look at lines parallel to the line of sight or optical axis. By taking only those lines which tangentially intersect the object surface and are parallel to the view direction (e.g., the y axis), and intersect them with the plane perpendicular to the view direction (in this case, the $x - z$ plane), we obtain the occluding contour of the object. These lines, called *contour generating lines*, form an infinite volume which encloses the object. By intersecting the volumes generated for each view, the bounding volume of the object may be obtained, and by using common raster lines to segment each contour, this bounding volume is split into parallelograms. These parallelograms are then themselves rasterized to obtain the line segments which form the volume segment representation used here.

Potmesil (Potmesil 1987) discusses the generation of octree models of 3-D objects from silhouettes. Intensity images are captured from multiple views. These images are then thresholded in order to isolate the objects from the background, using a threshold level determined from a histogram of the image, and thus obtain silhouettes. (The assumption is made that high contrast exists between the objects and the background.) The 2-D silhouettes are then converted into 3-D conic volumes: using the 4x3 perspective transform matrix for the camera (computed by camera calibration), recursively project each octree node into the image plane (thereby producing a 2-D "image" of the octree node cube) and determine if the octree node image (actually, its bounding rectangle) falls within the silhouette regions earlier determined. (If a node intersects both object and background regions, the eight leaf or child nodes of the parent branch node are then recursively considered.) This intersection of the octree node image with the image regions involves converting the image into a quadtree. The smallest quadtree node which encloses the octree node image is then recursively intersected with the octree node image to determine the contents of the octree node image. The result of this complete algorithm is a partial octree containing nodes which make up the conic volumes produced by the silhouettes obtained for one particular view. All partial octrees are then intersected, along with an octree designating the unseen volumes, to determine the complete model octree. One should mention that object concavities may not be represented using this algorithm.

Noborio and others (Noborio, Fukuda, and Arimoto 1988) present an algorithm for producing an octree representation of a workspace from multiple intensity images. This algorithm has a similar methodology as (Potmesil 1987) in making use of 3-D conic volumes, or "polyhedral cones", defined by the polygonal contour of the object image and the viewpoint for each view, and intersects these polyhedral cones using volume intersection. (Although it is not mentioned in the paper, it must be assumed that the perspective transform for each view is known in order to generate these cones.) First, the algorithm classifies each octree child node of a parent node as being inside, intersecting, or outside each polyhedral cone. This is accomplished by first checking points on the cubic region defining the node against the view cone surfaces. Intersecting nodes are subdivided into their eight child nodes and reclassified. Next, the algorithm intersects all of the view cones by recursively checking all "inside" nodes (from the first step) to see if they lie inside, outside, or intersect the "common region" (the volume intersected by all view cones). Nodes which are "inside" for every polyhedral cone are therefore "inside" the common region; those which are "outside" for any one cone are outside the common region. Again, intersecting nodes are subdivided and reclassified. Once the octree has been completely classified versus the common region, the nodes in the common region are output. Again, this algorithm may lose object concavities and an assumption must be made that each view contains all the objects or useful workspace.

Ahuja and Veenstra (Ahuja and Veenstra 1989) also generate octrees using silhouette images of an object. The images must be obtained from a subset of thirteen pre-determined viewing directions, severely limiting flexibility. In their experiments, they obtained simulated silhouette images of several geometric objects (generalized cones). For each image, depending on from which viewpoint in the subset it was viewed, an octree is generated. All of the octrees are subsequently intersected to produce a global octree.

Roth-Tabak and Jain (Roth-Tabak and Jain 1989) present an algorithm to generate a 3-D voxel-based environment model from simulated dense range images. Rather than simple binary (on-off) voxels, this algorithm uses voxels with three states: Void (off), Full (on), and Unknown (for voxels for which no meaningful information has yet been acquired). Starting with a model of completely Unknown voxels, the algorithm checks every non-void voxel within the scope of the sensor as follows:

1. The three-dimensional coordinates of each vertex for the voxel are found.
2. For each vertex, the pixel in the range image corresponding to the vertex's position (found by view-transforming the vertex coordinates) is identified.
3. If the distance of any vertex is smaller than the range of any corresponding range image pixel, the voxel is marked Void.
4. Otherwise, if the difference in range between the pixels corresponding to the vertices is within a threshold, the voxel is marked Full.

Thus, for each voxel to be classified, the algorithm makes up to eight range comparisons (one for each vertex of the voxel), in addition to view-transforming each vertex and thresholding. Obviously, this algorithm requires an extreme amount of processing for large arrays of voxels. (The size of array used in the paper is 64x64x16.) The reason for this is that this method assumes that each voxel occupies a finite volume. In comparison, most algorithms, including the two presented in this thesis, assume that each voxel is a single point in space (that point in space being the centroid of the voxel) and thus requires only a single range comparison to each reference point. (As we see in Chapter 4.3, the *direct mapping* method has one reference point, while the *peeling* method has two.)

Sharma and Scrivener's work (Sharma and Scrivener 1990) is very pertinent to my research. This paper discusses an approach for constructing 3-D object models using video (intensity) images from a scene (although no actual images are used for the paper). From these images, their approach involves deriving so-called "$2\frac{1}{2}$-D sketches". (They do not actually derive these images, but rather assume that such sketches can be accurately derived from images and obtain simulated $2\frac{1}{2}$-D sketches by creating a 3-D mathematical model of the geometry of objects in a scene.) The $2\frac{1}{2}$-D sketch is considered as an image with depth information for every pixel in the image – essentially, a completely-dense range image derived from a video or intensity image. With multiple $2\frac{1}{2}$-D sketches from different points of view, they introduce their 3-D model construction scheme – a "chipping" process which removes unwanted pieces from a block of voxels. The process sounds very similar to the "peeling" algorithm we propose, although without an explicit description of their process in the paper, comparison is difficult. However, we may compare topic areas: our project involves actual range data, rather than simulated $2\frac{1}{2}$-D sketches, and therefore must take into account the real-world aspects of the scanning system, e.g., separated camera and light source; we also use a linear octree to represent our data, thereby producing a more memory-efficient model; our project is application-oriented, its results required to be suited to path-planning, and thus is more concerned with modeling environments than objects.

Chapter 4

Range Image Integration

4.1 Description of System Components

The prototype system used for multiview range scanning and range image integration consists of four main components (see Figure 4.1):

- range scanner
- robot
- image capture computer
- image integration computer

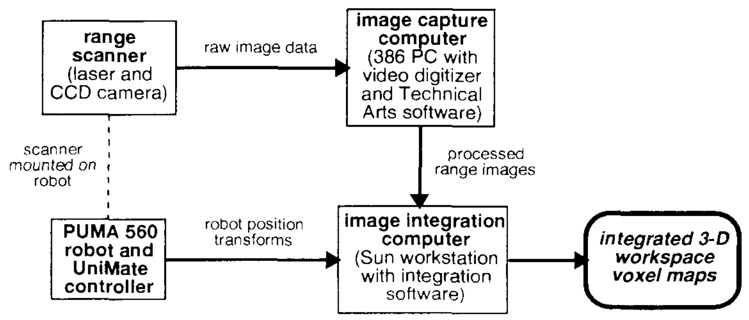

range scanner (laser and CCD camera)
raw image data
image capture computer (386 PC with video digitizer and Technical Arts software)
scanner mounted on robot
processed range images
PUMA 560 robot and UniMate controller
robot position transforms
image integration computer (Sun workstation with integration software)
integrated 3-D workspace voxel maps

Figure 4.1: Block diagram of our multi-view range scanning system.

The range scanner is a Technical Arts Corporation 100AT White scanner. The scanner itself has three subcomponents: a laser source, a camera, and a director module. The laser source currently being used is a helium-neon laser with an output of

10mW. This laser is admittedly somewhat more powerful than is required for my purposes, although the extra power may be useful in later extensions of this work – for instance, the scanning and modeling of larger workspaces. Spreader and director mirrors have been attached to the laser source to allow for area scanning (see Section 3.1). The second subcomponent of the scanner, the camera, is a Sony XC-75 CCD camera with a Schneider 10mm lens. The camera is especially sensitive to the laser light, thus capturing the image of the laser line intersecting objects in the workspace and transmitting the image to the image capture computer. The third subcomponent, the director module, contains the power supply for the laser and circuitry for control of the director mirror. The absolute accuracy of this scanner when properly calibrated is better than 1 part in 1000, or 0.1%. The accuracy of scanning and calibration is increased by taking multiple scans for each position of the laser line and averaging the range values obtained. Five scans are averaged for each laser line position during calibration, while fifty are averaged during scanning.

In order to use the scanner from multiple views, the scanner is mounted on a PUMA 560 robot, the second major component in the system. The scanner is mounted on a beam attached to the wrist joint of the PUMA (see Figure 4.2). Four degrees of freedom are available for movement of the scanner: the waist, shoulder, elbow, and one wrist joint. Programs and commands for controlling the PUMA are entered on an SGI workstation, which sends the appropriate signals to a separate PUMA controller. The PUMA may also be controlled by issuing commands on a teach pendant. From testing, PUMA positioning is accurate to approximately $\pm$ 1mm.

The image capture computer, the third major component of the system, is an 80386-based PC with a digitizer board for image capture. A second interface board in the computer controls the spreader mirror. Software on this computer operates the scanner, performs calibration, and calculates range values for the data points obtained with the digitizer. The scanned range images are stored in binary files on the computer's hard drive.

The binary range image files are then converted into ASCII data for use by the image integration computer, the fourth major component. The image integration routines are written for Sun workstations; however, one may remotely connect to a Sun through the SGI workstation which controls the PUMA. Because the image capture PC and the SGI workstation are currently separate, the ASCII range data

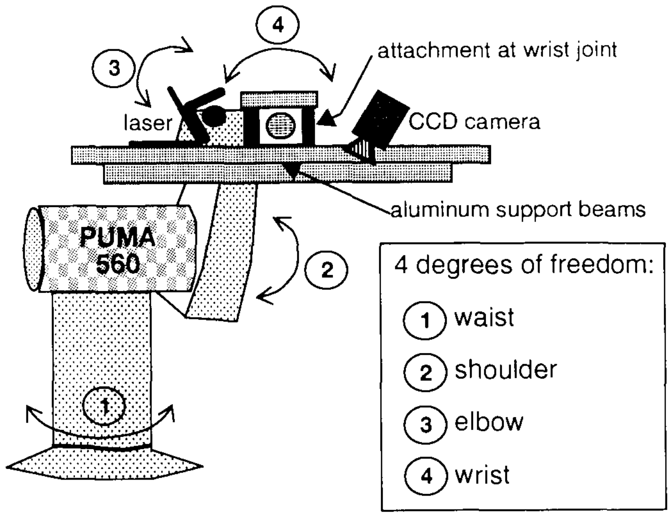


Figure 4.2: Mounting of the scanner on the PUMA wrist joint.

must be ported to the workstation by floppy disk. In the future, however, the two computers may be connected by an interface such that the range scanner may be controlled on the SGI and range data may be directed fed into the SGI.

4.2 Relations of System Transforms

4.2.1 Notation

This section describes the relationships of the various geometric transforms between different components of the range scanning and image integration system and different positions of the scanner and PUMA. The notation used in this thesis for geometric transforms is as follows: Assume we have a base reference frame, **P** (see Figure 4.3). If we wish to express the transform to obtain a second frame **S** relative to **P**, the notation would be: ${}^{\mathbf{P}}_{\mathbf{S}}T$.

Additionally, a point X may be expressed relative to frame **S** with the following notation: ${}^{\mathbf{S}}X$. We may then obtain the coordinates of point X expressed with respect to the base reference frame **P**: ${}^{\mathbf{P}}X = {}^{\mathbf{P}}_{\mathbf{S}}T\,{}^{\mathbf{S}}X$

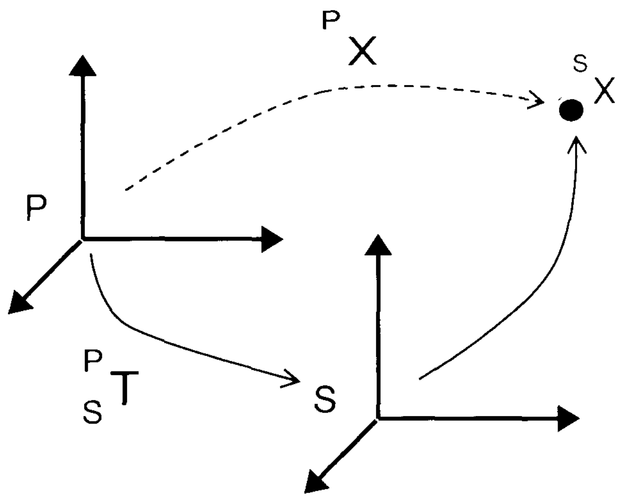


Figure 4.3: Notation used for designating geometric transforms.

4.2.2 Calibration of the Laser Scanner

In relating the transforms of the system, we must consider two separate issues: calibrating the laser scanner, and relating the PUMA to the scanner. Calibrating the laser scanner allows us to obtain accurate range measurements in the range images. As we shall see later, scanner calibration need only be performed once for a given set of multiple view range images, provided the positions of the laser and camera are not changed with respect to each other.

The laser scanner involves three separate components: the laser unit, the CCD camera, and the calibration gauge supplied by Technical Arts. When these three components are properly aligned and certain parameters (laser angle, camera angle, laser height, camera height, and gauge dimensions) entered into the control program, a calibration routine is executed. Briefly, the routine analyzes the image of the laser line on the calibration gauge and, based on the parameters entered and the expected shape of the calibration gauge, fixes the location of the range image reference frame (which we shall call $\mathbf{G_0}$. As illustrated in Figure 4.4, the $\mathbf{G_0}$ frame is fixed on a corner of the gauge (when the edge of the gauge is lined up with the first scan line); the z-axis is vertical, the y-axis is along the direction of the scan line, and the x-axis perpendicular to the scan line. As the scan line is stepped in the x-direction, the calibration routine uses the shape of the image to determine the correct (x, y, z) coordinates for each step.

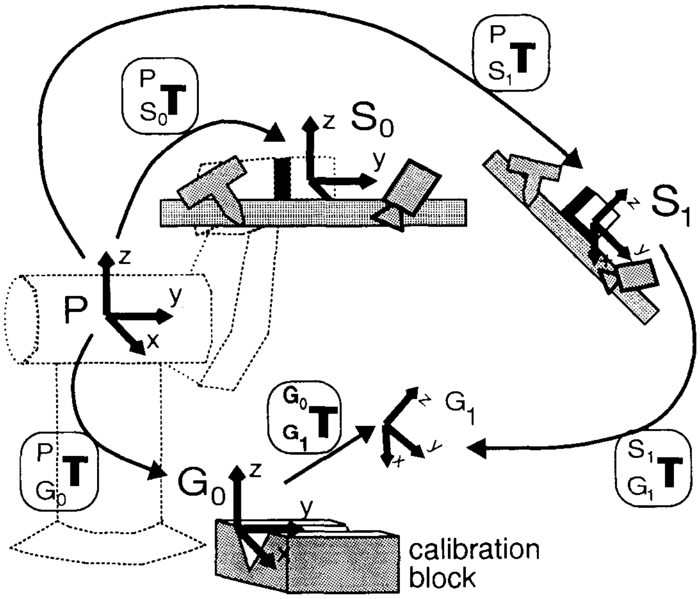


Figure 4.4: The transforms used in the scanner system.

4.2.3 Relating the PUMA to the Scanner

We can change our view by changing the position of the PUMA arm, and at each new position, we can obtain the transformation at that position from the PUMA controller. This transformation consists of the X, Y, and Z coordinates and the 3 rotational components of a tool reference point. These coordinates and rotations are with respect to the PUMA's base reference frame, which we'll call **P**. Let's call the tool transformation at position i, ${}^{\mathbf{P}}_{\mathbf{S_i}}T$. Now, let's say the transformation at the initial position, where the laser scanner is calibrated, is ${}^{\mathbf{P}}_{\mathbf{S_0}}T$. In other words, at the transformation ${}^{\mathbf{P}}_{\mathbf{S_0}}T$, the image exists in the base frame $\mathbf{G_0}$. However, we also need to establish relationship between the PUMA position and image space in order to locate the range image information with respect to the PUMA. This is the second issue in relating system transforms – linking PUMA to the physical space that has been related to the image space by laser scanner calibration, and in so doing, linking PUMA to the image space.

To accomplish this, we have considered a number of alternatives. A rough estimate should be attainable by manually measuring the translation and orientation difference of the point on the scanner calibration gauge corresponding to the origin of $\mathbf{G_0}$, with respect to the base of PUMA, where the base frame of PUMA, **P**, resides. This locates $\mathbf{G_0}$ in the global reference frame that is located at the base of the Puma, and so links image space to the global frame. Another more accurate method would involve measuring the relationship between a number of separate points on the gauge, obtaining the relationships by touching them with the PUMA moving a pointer of known translation from the tool transformation reference point and doing an error-fit to determine the best transformation to the origin of $\mathbf{G_0}$. Let's assume that, through one of these methods, we obtain the relationship of $\mathbf{P}$ to $\mathbf{G_0}$, which we'll call ${}^{\mathbf{P}}_{\mathbf{G_0}}T$. A point in frame $\mathbf{G_0}$ can be expressed in terms of the frame $\mathbf{P}$ by transforming it by ${}^{\mathbf{P}}_{\mathbf{G_0}}T$. Thus, for a point ${}^{\mathbf{G_0}}X$ in the space defined by $\mathbf{G_0}$,

$$ {}^{\mathbf{P}}_{\mathbf{G_0}}T \; {}^{\mathbf{G_0}}X = {}^{\mathbf{P}}X $$

where ${}^{\mathbf{P}}X$ is the point X expressed in the $\mathbf{P}$ frame.

We should next mention how these relationships are used for integrating multiple views. We obtain a second view by moving the PUMA to a position $\mathbf{S_1}$, with transformation ${}^{\mathbf{P}}_{\mathbf{S_1}}T$. The second range image we obtain would be in a new image space, with frame $\mathbf{G_1}$. In order to relate the image points in $\mathbf{G_1}$ back to the $\mathbf{G_0}$ frame, we must find the correct transformation ${}^{\mathbf{G_0}}_{\mathbf{G_1}}T$. If we look at Figure 4.4, we see that this transform can be obtained by following the transform path

$$ {}^{\mathbf{G_0}}_{\mathbf{G_1}}T = {}^{\mathbf{G_0}}_{\mathbf{P}}T \; {}^{\mathbf{P}}_{\mathbf{S_1}}T \; {}^{\mathbf{S_1}}_{\mathbf{G_1}}T $$

Note that with the scanner at $\mathbf{S_1}$ using the same scanner calibration as that for the scanner at $\mathbf{S_0}$, the geometry between the scanner position, i.e., $\mathbf{S_i}$, and the image frame $\mathbf{G_i}$ must remain constant. In other words,

$$ {}^{\mathbf{S_0}}_{\mathbf{G_0}}T = {}^{\mathbf{S_1}}_{\mathbf{G_1}}T $$

Thus,

$$ {}^{\mathbf{G_0}}_{\mathbf{G_1}}T = {}^{\mathbf{G_0}}_{\mathbf{P}}T \; {}^{\mathbf{P}}_{\mathbf{S_1}}T \; {}^{\mathbf{S_0}}_{\mathbf{G_0}}T $$

We can express this in terms of measurable transforms. ${}^{\mathbf{G_0}}_{\mathbf{P}}T$ is directly measurable (see Appendix A), while ${}^{\mathbf{P}}_{\mathbf{S_1}}T$ is provided by the PUMA controller. The relationship

${}^{\mathbf{S_0}}_{\mathbf{G_0}}T$ can be obtained from the transformation for $\mathbf{S_0}$ from PUMA and the measured relationship between $\mathbf{P}$ and $\mathbf{G_0}$:

$$ {}^{\mathbf{S_0}}_{\mathbf{G_0}}T = {}^{\mathbf{S_0}}_{\mathbf{P}}T \, {}^{\mathbf{P}}_{\mathbf{G_0}}T $$

$$ {}^{\mathbf{S_0}}_{\mathbf{G_0}}T = {}^{\mathbf{P}}_{\mathbf{S_0}}T^{-1} \, {}^{\mathbf{P}}_{\mathbf{G_0}}T $$

So, in terms of measurable transforms,

$$ {}^{\mathbf{G_0}}_{\mathbf{G_1}}T = {}^{\mathbf{G_0}}_{\mathbf{P}}T \, {}^{\mathbf{P}}_{\mathbf{S_1}}T \, {}^{\mathbf{P}}_{\mathbf{S_0}}T^{-1} \, {}^{\mathbf{P}}_{\mathbf{G_0}}T $$

Using ${}^{\mathbf{G_0}}_{\mathbf{G_1}}T$ then, we can relate a point ${}^{\mathbf{G_1}}X'$ in the image frame $\mathbf{G_1}$ into the corresponding point ${}^{\mathbf{G_0}}X'$ in the image frame $\mathbf{G_0}$, and finally into point ${}^{\mathbf{P}}X'$ in the PUMA base frame $\mathbf{P}$:

$$ {}^{\mathbf{G_0}}X' = {}^{\mathbf{G_0}}_{\mathbf{G_1}}T \, {}^{\mathbf{G_1}}X' $$

$$ {}^{\mathbf{P}}X' = {}^{\mathbf{P}}_{\mathbf{G_0}}T \, {}^{\mathbf{G_0}}X' = {}^{\mathbf{P}}_{\mathbf{G_0}}T \, {}^{\mathbf{G_0}}_{\mathbf{G_1}}T \, {}^{\mathbf{G_1}}X' $$

This system is expressed pictorially in Figure 4.4. (Note that the X and Y axes on the scanner calibration gauge frame $\mathbf{G_0}$ have been swapped from the X and Y axes used by the scanner in order to maintain consistency in orientation with the PUMA $\mathbf{P}$ frame.)

Because we have the relationship of $\mathbf{P}$ to $\mathbf{G_0}$, i.e., ${}^{\mathbf{P}}_{\mathbf{G_0}}T$, we can relate images in $\mathbf{G_1}$ (and images in subsequent views in image frames $\mathbf{G_i}$) to the global reference frame $\mathbf{P}$, simply by transforming images in $\mathbf{G_i}$ back to $\mathbf{G_0}$ by calculating ${}^{\mathbf{G_0}}_{\mathbf{G_i}}T$ for each $\mathbf{G_i}$.

4.3 Image Integration Methods

4.3.1 Overview

A number of factors characterize the range image integration problem presented in this thesis:

1. **Transform availability**: The geometric transforms between each view (or between each view and the base reference frame) are available, thus providing a simpler problem (and more accurate solution) than if transforms needed to be derived based on the sensed range data.

2. **Voxel-map integration**: The range images are integrated directly into a voxel map, unlike some approaches which segment range images in order to obtain a geometric model.

3. **Two-component scanner**: As explained earlier, our scanner is a two-component system, with each component in a separate location. Many other range scanners have both components at the same or nearly the same location. This factor is particularly important when the peeling algorithm is discussed later.

4. **Path planner suitability**: The prime focus of the range image integration project is the production of a model suitable for use by a path planner. Because the path planner does not care about the geometric properties or finer details of the objects in the workspace, we need not have an exact representation of each object; merely their spatial occupancy (i.e., size and location in space) need be accurate. However, the objects should be represented as "filled-in" or solid volumes rather than surface shells, in order to avoid non-collision positions within objects. In addition, unscanned areas must be treated as obstacles: without knowledge of the contents of these areas, the planner should avoid them.

With these factors in mind, two separate integration algorithms were investigated: (i) *direct mapping* and (ii) *peeling*. The two methods, which are explained in detail in this chapter, differ in the way they interpret the range data in each image and in the nature of the resulting integrated models produced by each method. As we shall see, the peeling algorithm produces a model more suitable for a path planner, though the direct mapping algorithm is useful for verification of correctness of the transforms.

4.3.2 Direct Mapping

The *direct-mapping* algorithm is so named because it maps pixels in range images directly into our voxel map (3-d bitmap). In other words, if a pixel in a range image contains depth information, that depth information is used to determine the location of one (and only one) voxel in the voxel map – that voxel being on the surface of an object or obstacle in the workspace. The direct mapping algorithm thus creates a surface-map model of the workspace.

Below is a high-level algorithm for direct mapping (calculation of PUMA-to-image-frame transforms was explained in the Calibration chapter):

```
Main routine: Direct_Map
Input:
  • array of images $Image[]$
  • number of images $Num\_Images$
  • array of matrices Transform = PUMA-to-image-frame transform for each image ($^{\mathbf{P}}_{\mathbf{G_i}}T$)
Output: Voxel array representing workspace $VoxArray$.

  1. Initialize all voxels in $VoxArray$ to OFF
  2. For $n = 1$ to $Num\_Images$ do:
     (a) Search incrementally in $x$ and $y$ (relative to range image frame) in range image $Image[n]$
         for a pixel with a valid range value ($z$)
     (b) For each such pixel, do:
         i. Let $Vector = [x, y, z]$ be the range image pixel converted to a vector in the range
            image frame
         ii. Let $PUMA\_Vector = \mathbf{Transform[n]} \times Vector$ be the vector transformed into the
             PUMA frame
         iii. Locate the voxel in $VoxArray$ corresponding to $PUMA\_Vector$ and set it to ON
         iv. Evaluate next pixel in $Image[n]$ (go to 2a)
     (c) When all pixels evaluated, go to next range image (go to 2)
```

In the voxel map, the direct mapping algorithm produces thin shells, a few voxels thick, that represent surfaces scanned in the range images. By viewing the results of direct mapping, the correctness of the "fitting" of the multiple-view range images can be determined – if correct, one should be able to distinguish the objects that were scanned. However, the objects are not represented in a solid fashion. The object surfaces (if scanned) have representation in the voxel map as thin shells, but the interior of the objects are not represented. In addition, the areas that have not been scanned in our range images (the "unknown" areas) are not mapped to the voxel map; hence, these areas are considered to be empty space rather than as obstacle areas to be avoided. These two drawbacks indicate that the direct mapping algorithm is unsuitable for generating models for path planning.

4.3.3 Peeling

The *peeling* algorithm is an attempt to overcome the drawbacks of direct mapping. The concept of peeling is similar to that of woodcarving: starting with a completely filled voxel map (solid block of wood), range images are used to peel away voxels known to be in free space (chip away wood), so that the remaining voxels represent the workspace model. The underlying basis for the peeling algorithm is that each range pixel (i.e., a range image pixel that has a range value) provides two pieces of information: (i) the location of a point *on* an object's surface, and (ii) that points in space between that surface point and the laser, and between the surface point and the camera, are free: a scanned point corresponding to the range pixel must have had the laser beam hit it and must have been seen by the camera. Therefore, by starting with a completely voxel-filled workspace model and, for each scanned surface point, removing voxels from the model along the vector from the surface point to the laser, as well as the vector from the surface point to the camera, an accurate workspace model is achieved (see Figure 4.5). The positions of both the camera and the director mirror on the scanner must be measured relative to some known point (e.g., the robot's tool center point). Note that range image pixels with no range value (i.e., unscanned points) are not mapped to the voxel array; thus unscanned areas are left unpeeled and treated as obstacle areas.

Below is a high-level algorithm for peeling:

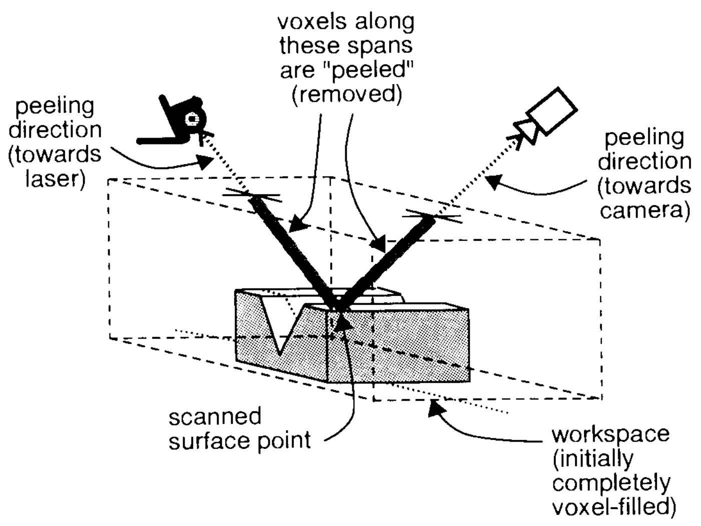

voxels along these spans are "peeled" (removed)
peeling direction (towards laser)
peeling direction (towards camera)
scanned surface point
workspace (initially completely voxel-filled)

Figure 4.5: The peeling technique for range image integration.

Main routine: **Peeling**

Input:

- array of images *Image*
- number of images *Num_Images*
- array of matrices **Transform** = PUMA-to-image frame transform for each image (${}_{\mathbf{G}_0}^{\mathbf{P}}T$)
- locations of the laser *LaserPos* and camera *CamPos* with respect to image frame

Output: Voxel array representing workspace *VoxArray*.

```
1. Initialize all voxels in $VoxArray$ to ON
2. For $n = 1$ to $Num\_Images$ do:
    (a) Let $LaserPosT$ = Transform[n] $\times$ $LaserPos$ be the transformed laser position relative
        to PUMA frame
    (b) Let $CamPosT$ = Transform[n] $\times$ $CamPos$ be the transformed camera position relative
        to PUMA frame
    (c) Search incrementally in $x$ and $y$ (relative to range image frame) in range image $Image[n]$
        for a pixel with a valid range value ($z$)
    (d) For each such pixel, do:
        i. Let $Vector = [x, y, z]$ be the range image pixel converted to a vector in the range
           image frame
        ii. Let $PUMA\_Vector$ = Transform[n] × $Vector$ be the vector transformed into the
            PUMA frame
        iii. Let $LaserPeelDir = LaserPosT - PUMA\_Vector$ be the peeling direction towards the laser
        iv. For all voxels between the endpoint of $PUMA\_Vector$ and $LaserPosT$ in direction
            $LaserPeelDir$:
            A. Turn voxel OFF
        v. Let $CamPeelDir = CamPosT - PUMA\_Vector$ be the peeling direction towards
           the camera
        vi. For all voxels between the endpoint of $PUMA\_Vector$ and $CamPosT$ in direction
            $CamPeelDir$:
            A. Turn voxel OFF
        vii. Evaluate next pixel in $Image[n]$ (go to 2c)
    (e) When all range pixels evaluated, go to next range image (go to 2)
```

The peeling technique is highly dependent on the choice of views of each scan: in order to effectively remove "noise" voxels, the views must encompass as much of the scene as possible. Hence, multiple scans of the same surface (from different points of view) may be required in order to reduce noise and increase the effectiveness of this algorithm (the direct-mapping algorithm requires only a single scan of a surface – with data of reasonable density – in order to represent that surface). As the number of scans is increased, processing time is likewise increased, as is memory usage. However, the resulting voxel map is suitable for path-planning applications in that objects have a solid representation and unknown areas are represented as obstacle space.

4.4 Experiments

For the experiments, a number of objects (shown pictorially in Figure 4.6) were placed in the robot workcell and scanned from multiple views using the range scanner system described earlier. The range images obtained are shown in Figure 4.7. Using software on the Sun workstation platform, the range images were integrated using both the direct-mapping and peeling algorithms; voxel map files were generated in both cases. The voxel maps were viewed on an SGI workstation; screen snapshots were taken for both the direct-mapping and peeling results and are shown in Figure 4.8 and Figure 4.9 respectively. Each voxel map has been framed by a wire-frame cube to show, generally, the 3-d orientation of the voxel map. The dotted areas in each snapshot indicate voxel-mapped areas, i.e., obstacle areas.

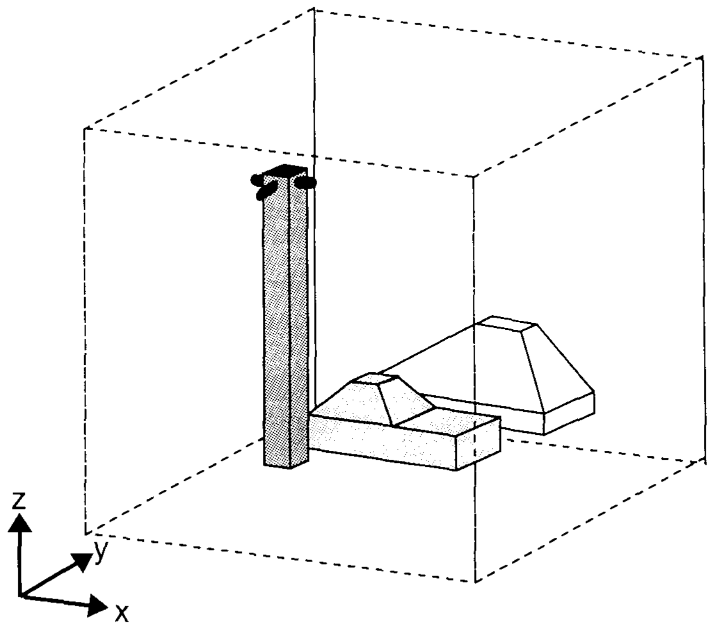


Figure 4.6: A schematic of the objects used in the experiments. Axes are used as reference for voxel map results only.

The results for direct mapping (Figure 4.8) show the correct shape and position of the objects. However, as stated previously, the voxel maps generated by the direct mapping algorithm are shell representations of the objects in the workcell. Such representations may be useful in some circumstances, but if proper solid representations are required, the raw results of direct mapping cannot be used. A possibility is to solidify these shell representations using some sort of fill algorithm starting from their center, but other problems arise, the most significant of which being that gaps in the range data (due to occluded surfaces, unscanned surfaces, or surface properties which prohibit or limit scanning) would cause such a fill algorithm to function incorrectly. A much greater shortcoming of this technique, however, is that one cannot tell which areas are unscanned simply by looking at the voxel map. For a path planner, these areas should be avoided (i.e., treated as obstacle, not as free space) because their contents are unknown – these unscanned areas may contain objects which are unknown to the planner.

The results for the peeling algorithm (Figure 4.5) show the two advantages of this algorithm over direct mapping: that objects have a solid representation, and that unscanned areas are mapped as obstacles. They also, however, show a possible problem: that without a sufficient number of range scans of wide enough field of view, large numbers of extraneous voxels are left in the voxel map, enough to obscure the actual objects in some cases. (Of course, using 2-D images to depict a 3-D environment – especially one being represented as individual voxels – results in even greater obscurity.)

The advantages of peeling – that is, representation of object areas as solids and representation of unscanned areas as obstacles – are extendable to situations where a greater number of range images of wider range may be obtained and integrated using more powerful hardware than was available for this thesis (and more readily available today). Obviously, the resulting voxel maps for these situations would contain far fewer extraneous voxels and thus higher definition of object areas (lower obscurity).

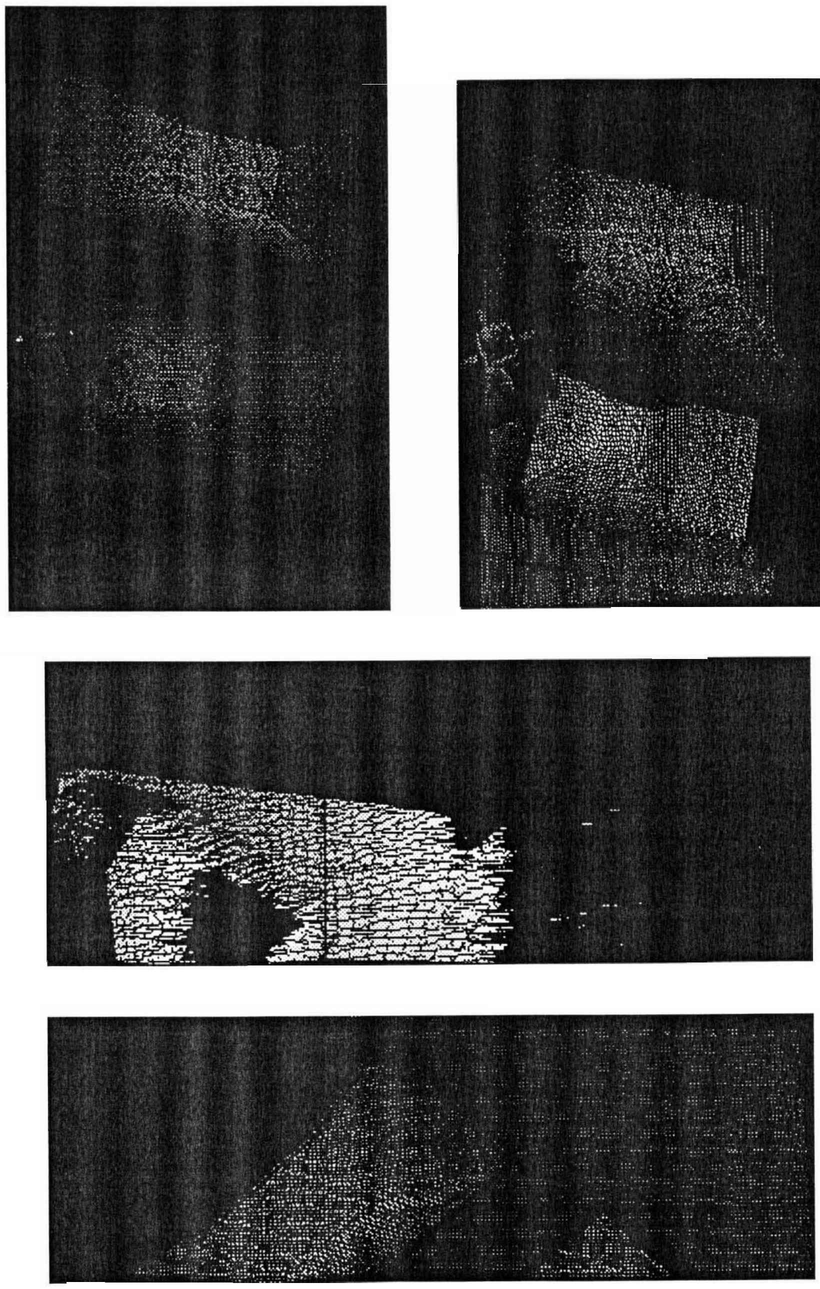
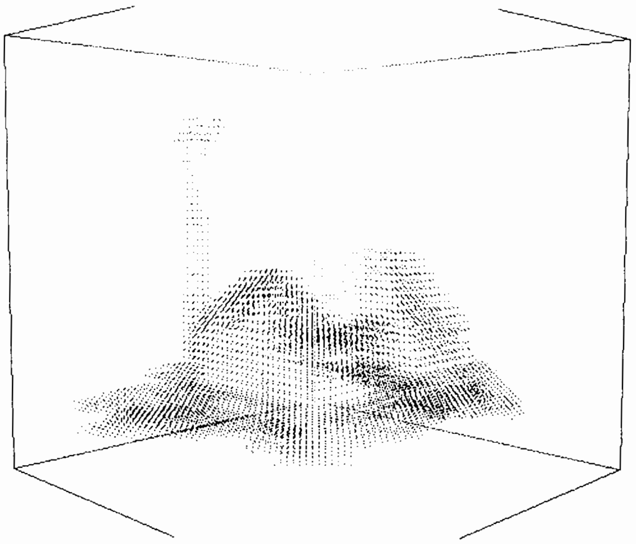
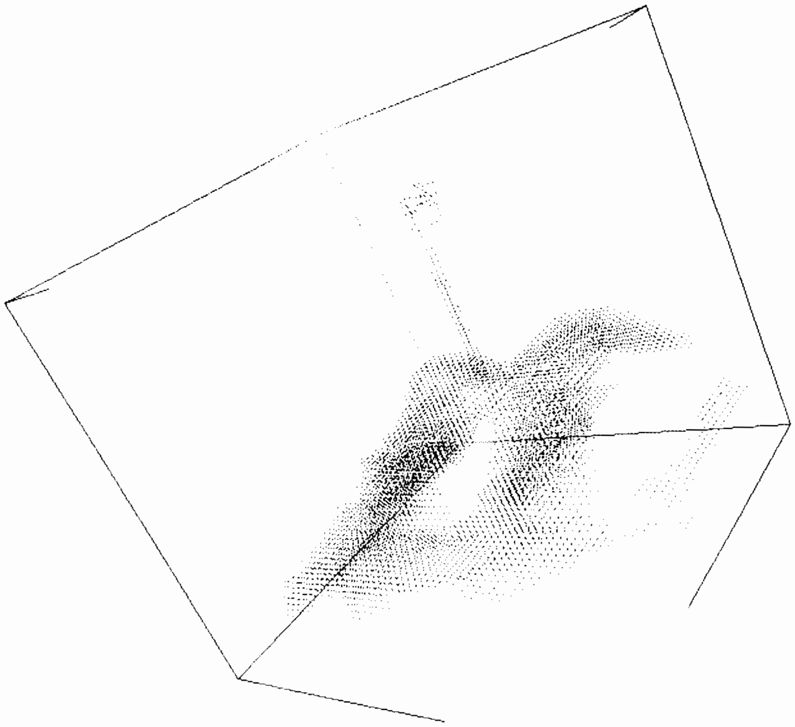
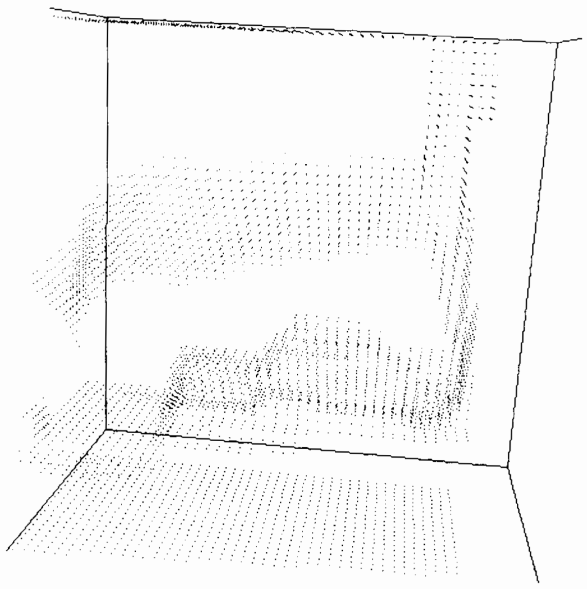


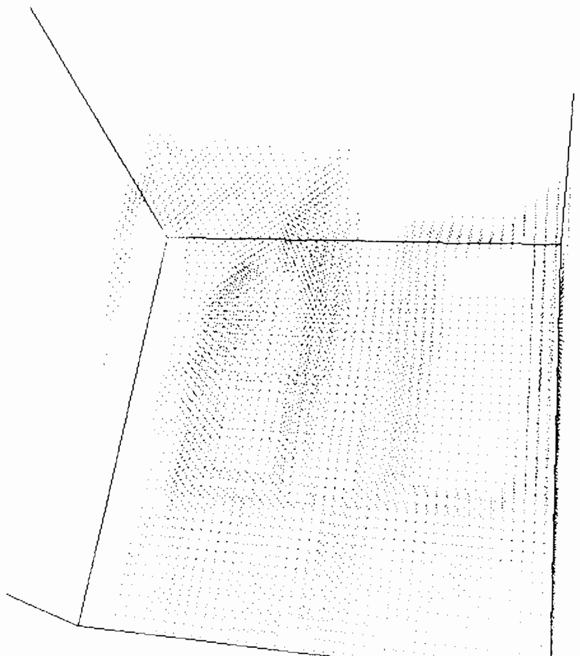
Figure 4.7: Range images used for integration.


$+x, -y$ cube planes closest to viewer


$+x, +y$ cube planes closest to viewer



+y cube plane closest to viewer



+z cube plane closest to viewer

Part II

Hierarchical Octree-Based Distance Map Representation for Path Planning

Chapter 5

Introduction

Distance maps (also known as distance transforms or distance fields) or their variations, such as potential fields, have been used in robotics for a variety of path planning and collision-avoidance applications (Latombe 1991; Jarvis 1993). Computational implementations of these distance maps invariably involve discretized or grid-based representation of the domain over which the distance map is defined. Consider, for instance, a discretized distance map that maps the (L_1 or Manhattan) distance in voxels from a particular voxel to the nearest obstacle. This distance map is represented in an array with each voxel containing an integer representing the distance between it and the nearest obstacle voxel. An example of a 2-D pixel-level distance map is shown in Figure 5.1. One of the many uses for such distance maps has been for efficient collision-detection and path planning in static environments. For example, assuming a spherical robot model (often used for mobile robots), collision checking can be done very efficiently and easily if such a distance map has been pre-generated for the workspace. It merely involves comparing the radius of the robot to the value of the distance map at the (x, y) location of the robot. Using spherical representations as in (del Pobil, Serna, and Llovet 1992) for the entire manipulator arm, this basic collision-detection computation can then be carried out for each sphere in the manipulator to compute the collision situation for the entire manipulator (Greenspan and Burtnyk 1996). This scheme forms the core collision detection component of a path planner developed in (Yang, Gupta, and Greenspan).

A main drawback of such discretized representations, however, is the large amount of memory required to store it. For instance, for a three-dimensional space measuring

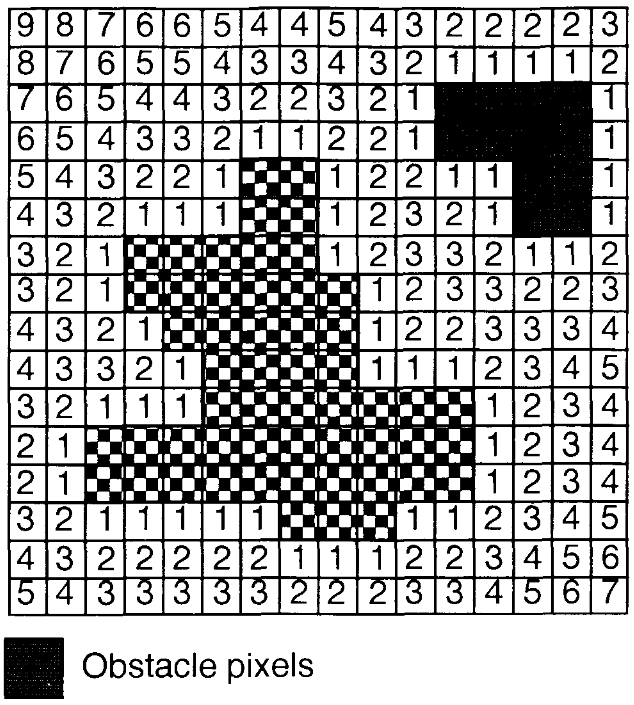


Figure 5.1: Example of a pixel-level Manhattan distance map.

1000 units per side, and assuming each number in the distance map is a two-byte integer, one would require $1000^3 \times 2 = 2$ gigabytes of memory. Larger spaces and/or finer resolution, of course, increase this usage exponentially. In the image processing field, a more memory-efficient method than a raw binary array is the octree data structure[1] (Samet 1990a). The octree representation is a "binary" representation of space, where the space is recursively subdivided into hierarchically represented cells as nodes in a tree. The nodes are labeled *white* or *black* indicating if a node is completely free or completely occupied, respectively. In effect, a single *white* or *black* node can be substituted for a great number of binary array elements while requiring only a comparatively tiny fraction of memory, hence the greater memory efficiency. A third category, *grey* nodes, represent cells which are partially occupied. *There is, however, no distance information associated with each node of an octree.* Furthermore, although octrees have been used in robotics for collision detection ((Hayward 1986), (Arimoto,

[1]Commonly called quadtrees in 2-d and octrees in 3-d. Because of the emphasis here on 3-D modeling and representation, we shall use the term *octree* in situations where either 2-D and 3-D representations may be involved.

Noborio, Fukuda, and Noda 1988)), the essence of previous approaches is to detect if a given point lies inside the black node of the environment octree. *There is no known previous attempt using octrees that incorporates the use of distance information in the collision detection process.*

This part of the thesis proposes a novel hierarchical method for representing distance maps, called the *Octree Distance Map* or ODM. As the name indicates, the ODM representation adapts and augments the conventional octree data structure in order to represent distance maps in a hierarchical manner. The ODM representation drastically reduces the expensive memory requirements associated with a voxel-array based distance map – by more than an order of magnitude – with some trade-off in collision-detection computations. Additionally, the approach presented here improves substantially on the collision-detection performance of conventional octrees. The results of this thesis illustrate the advantageous compromise achieved: although an ODM requires slightly more memory than an unaugmented octree for the same workspace (though still much less memory than a voxel-based distance map structure), the ODM provides a significant improvement in performance over an octree for collision detection. Figure 5.2 qualitatively illustrates this compromise.

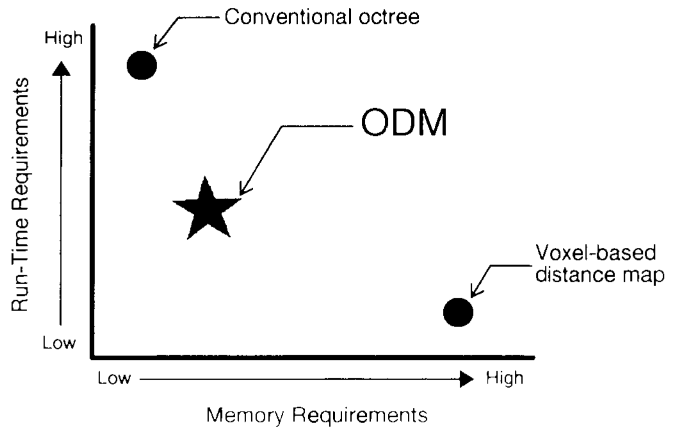


Figure 5.2: The memory-speed compromise achieved with the ODM, illustrated qualitatively.

Two algorithms are presented here: (i) given a conventional octree that represents a given workspace, build an ODM that hierarchically represents the distance from the obstacles, and (ii) given such an ODM, compute whether a spherical robot is in collision with the obstacles if positioned at a given point in workspace. The ODM building process is off-line and is executed only once for a given workspace, precomputing the ODM for multiple collision detections. The ODM collision detection process may be repeatedly performed, and thus is required to be efficient. In addition, the robot radius size may vary between separate executions of the collision detection algorithm, using the same ODM; we cannot assume a fixed radius length. Although the main motivation has been collision detection, the algorithm can be easily modified for determining distance between a robot and its environment.

Chapter 6

Background

6.1 Previous work

(Samet 1990a) discusses the issue of representing distance hierarchically and defines a *distance transform* for quadtrees. This distance transform represents the shortest distance from the center of each obstacle (*black*) node to a boundary between a *black* node and a *white* node. An algorithm is given for computing the L_∞ (Chessboard) distance transform for a quadtree. Briefly, the algorithm searches the quadtree for *black* nodes in top-down traversal. For each *black* node, its eight neighbours are examined in order to determine the distance to the closest *white* piels to the *black* node. (Shneier 1981) offers a similar distance transform for quadtrees (using the L_1 distance), with the addition of storage of the minimum distance to a *white* pixel in each of the four neighbour directions (north, south, east, and west).

Although the central issue of this thesis, that of representing distance hierarchically, is the same as in the above work, the main motivation was very different: that of efficient image representation as opposed to collision detection. Therefore, there are significant differences between the ODM representation and the above work. In particular, the use of a single distance value for an octree node, while adequate for image representation, is inefficient for collision detection (see Section 7.1). For greater collision detection efficiency, therefore, an ODM associates a minimum-maximum distance range with each *white* node of the octree. Building an ODM, consequently, is more search intensive, and hence time consuming. However, it leads to more efficient collision detection. Additionally, this distance range is very compactly represented in the ODM. The ODM maps distances of free-space nodes, not obstacle nodes. Finally, we present efficient collision detection algorithms based on an ODM representation of a static environment. A distinguishing feature of these algorithms is the use of hierarchical distance information which, to the best of my knowledge, has not been done before in collision detection algorithms based on octrees.

In other work, (Hayward 1986) outlines two approaches. The first approach assumes a robot representation where each volumetric piece of the robot is represented by a cylinder with a hemisphere on each end. An octree is duplicated for each represented robot cylinder and hemisphere, and obstacles are grown by the radii of the component volumes. The robot is then represented with line segments, and each segment is recursively checked (by binary subdivision) for collision in the appropriate octree. Obviously, this approach is extremely memory-intensive, requiring storage of multiple octrees. Without growing multiple octrees, the 3-d models of each robot link would need to be converted into octrees and subdivided. The second approach uses only a single octree, but its methodology is less robust: the robot is represented as a number of selected control points based on the robot's boundary surface representation, and each control point is located within the octree to determine if it is within an obstacle (thus, collision). Questions arise as to the spacing and position of the control points, and the number of such points, required in order to ensure proper execution. In addition, if an object were to be completely within the robot's boundary representation, the algorithm may not detect any interference between the object and the control points.

(Arimoto, Noborio, Fukuda, and Noda 1988) has also used conventional octrees for interference detection and path planning. The proposed approach to collision detection is to subdivide the space containing robot links into octree nodes and evaluate each node to determine if it contains obstacle regions, robot regions, or both. If the node contains both, it is recursively subdivided until its components are classified to be either completely inside or completely outside a robot, an obstacle, or both. A collision results if a node is in both the robot and an obstacle. In effect, this approach intersects an octree model of the robot with an octree of the workspace. The main disadvantage here is the high degree of computation involved - a new octree must be recomputed for each new robot configuration.

Another related body of work is (Noborio, Naniwa, and Arimoto 1990) which proposes a quadtree-based algorithm for path-planning for mobile robots. Computationally, collision detection is simplified by using a quadtree whose nodes are no smaller than the size of the robot; any node containing any part of an obstacle is considered *black* – in other words, *grey* nodes are eliminated from the quadtree. Thus, after locating the node in which the robot is located, the robot is known to be in collision if the robot is in a *black* node, and not in collision if it is in a *white* node. While this approach to collision detection is simple and not computationally expensive, and well-suited for its stated path-planning problem in 2-D, a number of drawbacks exist for the use of this approach in collision detection. The greatest disadvantage is that this approach is overly conservative. For instance, because a *grey* node is considered *black* even if it contains a single obstacle pixel, a robot placed in close proximity to the real obstacle will most likely be considered in collision even if the robot is not. This problem would be amplified given a highly occupied workspace and/or a large robot, when any free space would likely be within "*black*" nodes.

It should be noted, at this point, that none of the references mentioned above use distance in their collision detection processes. This observation further separates these previous efforts from the work presented in this thesis, which makes significant use of distance in collision detection.

More up-to-date work in this field includes (Bandi and Thalmann 1995), which uses spatial subdivision to perform collision detection on animated rigid bodies, and (Egbert and Winkler 1996), which uses vector fields for path planning.

6.2 Octrees

Of the many forms of representation for spatial data, octrees are one of the most widely used (Samet 1990b). They provide much greater memory efficiency than raw binary arrays as well as a simple structure and good operational flexibility. A conventional tree structure is composed of a hierarchy of *nodes*. A node, in tree-structure terminology, is a structure which represents a section of space. The size of that section of space varies from node to node within the tree, from an element of the finest resolution (a *pixel* in 2-D or *voxel* in 3-D) to the entire space itself; the further a node is away from the tree's root, the smaller is its section of space. Each node in a conventional octree is given one of three states to represent the spatial content of that particular section of space: *black* indicates that the entire node is in obstacle space, *white* indicates that the node is free space (free of obstacle), and *grey* indicates a node with a mixture of free space and obstacle space.

Each node contains up to a certain number of pointers (four for quadtrees, eight for octrees) to *child nodes* or *subnodes*, each of which representing a section of the space represented by the original node. These subnodes may in turn have children, who may have children, and so on, all the way to the bottom of the tree, where each node corresponds to a unit spatial element (pixel or voxel). The number of children for a particular node depends on the colour or state of the node. *white* and *black* nodes have no children, while *grey* nodes have pointers to each of their *black* or *grey* child nodes. Figure 6.1 provides an example of the breakdown of a 2-D image into a quadtree.

Location codes are a concept commonly associated with linear or pointer-less octrees (Gargantini 1982; Samet 1990b) to identify a particular node in the octree. A location code consists of a string of octal digits (or the equivalent in decimal). Each digit provides information as to which branch, 0 to 7, of an octree to traverse at each level in order to locate the node. For example, the location code 3504_8 (or 1860_{10}) allows us to locate the node by following branch 3 of the root, then branch 5 of the node at level 1 (level 0 being the root), then branch 0 of the node at level 2, then finally branch 4 of the node at level 3.

Besides providing a convenient way of identifying octree nodes within a tree, location codes also encode Cartesian-coordinate information, and thus are a way of locating the Cartesian location of an octree node, and a way of finding the location of a set of Cartesian coordinates in an octree. Assume we use a *Morton ordering* of octants (see Figure 6.2) [1] . Given an octal-digit location code, we first convert each digit to its binary form, which is composed of three binary digits, b_x b_y b_x. By forming a binary string of all the b_z and converting back to decimal, we obtain the z-coordinate. We likewise obtain the y- and x-coordinates. For example, we convert the location code 3504_8 to binary: 011 101 000 100. For the z-coordinate, we have the following: $0101_2 = 5$. For y, we have $1000_2 = 8$, and for x, $1100_2 = 12$. To obtain

[1] A left-handed coordinate system has been implicitly assumed, with the z-axis pointing into the page. A trivial modification in the ordering – with nodes 0, 1, 2, and 3 forming the back plane and nodes 4, 5, 6, and 7 forming the front plane – can be used to be consistent with a right-handed coordinate system, with the z-axis pointing out of the page.

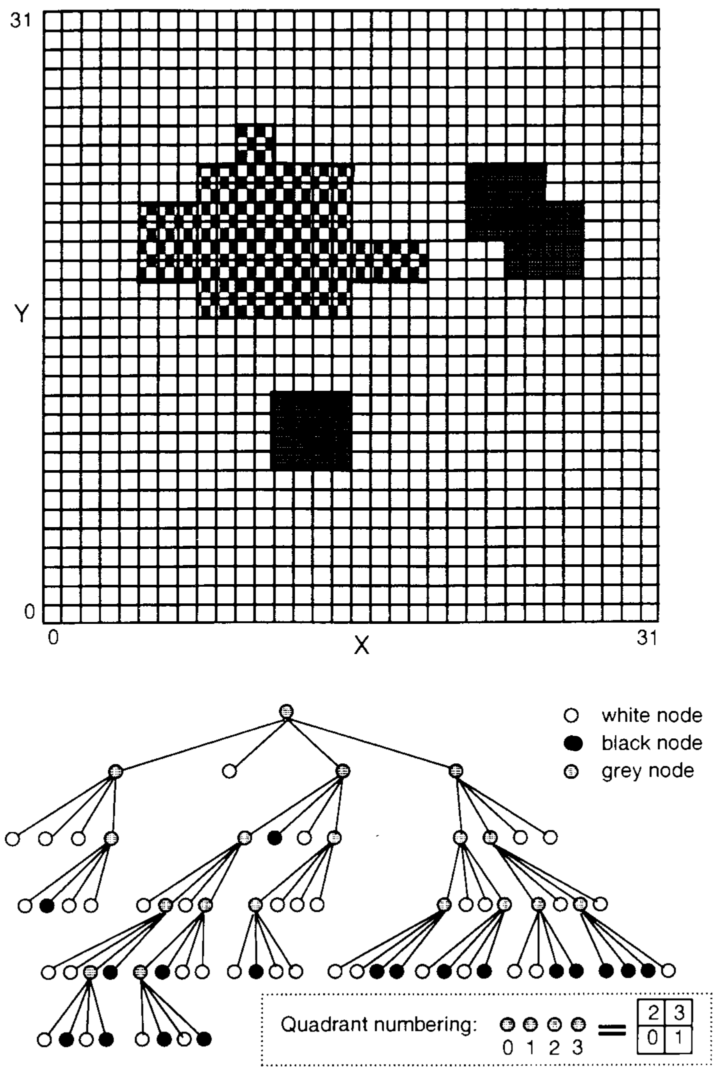


Figure 6.1: A 2-D image and corresponding quadtree.

a location code (and thus octree location) from (x, y, z), the reverse process is used.

Despite the use of conventional octree structures in this thesis rather than linear octrees in building and utilizing the ODM, location codes are used in three situations. The first situation arises in ODM construction when we wish to locate all nodes of level L which are at a (nodal separation) distance r from a given *white* node. In effect, we wish to determine the level-L nodes forming the surface of a *Manhattan sphere* of radius ($r\times$ size of level-L node). First, assume we have the location code LC for the *white* node W. (Note that LC refers to the voxel in W closest to the origin of space, i.e., $(0, 0, 0)$, assuming that our space is non-negative. The voxel called LC is

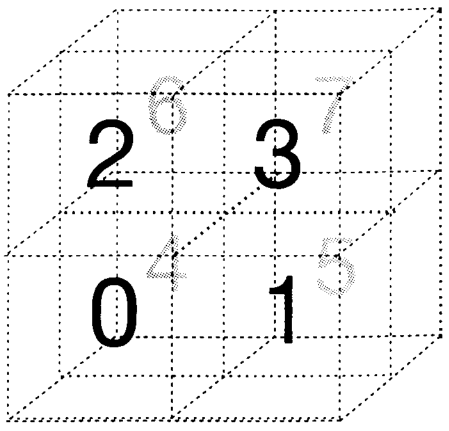


Figure 6.2: Octant numbering based on Morton ordering.

the *nodal reference voxel* of W. For the numbering scheme shown in Figure 6.2, the nodal reference voxel is the leftmost, bottom-most, minimum-z voxel in W.) From LC, we can then derive, as above, the Cartesian coordinates (x_0, y_0, z_0) of the nodal reference voxel of W. Let $s = 2^{(D-L)}$ be the length of one side of a node at level L. Then if T is a level-L target node at nodal distance r from W, its reference voxel is at $(x_n, y_n, z_n) = (x_0 + is, y_0 + js, z_0 + ks)$, and $(\mid i \mid + \mid j \mid + \mid k \mid) = r$. Therefore, by adding or subtracting increments of s to the coordinates of the reference voxel of node W, we obtain the reference voxel of T. Conversion of the reference voxel's coordinates to location code provides us the node T.

In the second situation, we are given the Cartesian coordinates of a robot for collision detection purposes, and must determine the octree node in which the robot is located. Here, the (x, y, z) coordinates are converted to binary, the binary digits are interlaced to obtain octal-digit location code, and the location code is used to traverse the octree to the desired node.

The third situation arises when we wish to determine the nodal separation between two nodes of level L given their location codes, LC_1 and LC_2. First, a property of location codes should be pointed out here: for a node of level $L < D$, where D is the depth of the octree, at least the last $D - L$ octal digits of the location code will be zero. This property is a result of the location of the nodal reference point discussed above. For $L = D - 1$, reference points will be in increments of 2 along each axis, for $L = D - 2$, the increment is 4, and so on. Thus, in order to determine the nodal separation between two nodes, we merely drop the proper number of trailing zeros $(D - L)$, find the two sets of Cartesian coordinates associated with the truncated location codes, and take their difference. The nodal separation can then be found by summing the differences in x, y, and z.

6.3 Distance

The distance function that maps a pair of points, p and q, into non-negative numbers is denoted by $d(p, q)$. The two common distance measures used in this part of the thesis are the L_1 (Manhattan) distance denoted by d_M, and the L_2 (Euclidean) distance, denoted by d_E. Unless Euclidean distance is specified, the Manhattan distance is used throughout this part of the thesis. For completeness, the distance calculations are shown here:

$$d_M = \| p - q \|_1 = \sum_{i=1}^{n} | p_i - q_i |$$

$$d_E = \| p - q \|_2 = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

where $p = p_1, \ldots, p_n$, $q = q_1, \ldots, q_n$, and n is the number of dimensions of space.

The distance d from a point p to a set X is $d(p, X) = min_q\{d(p, q), q \in X\}$. The distance between two sets, X and Y, is $d(X, Y) = min_p\{d(p, Y), p \in X\}$.

Chapter 7

Implementation of Octree Distance Maps

7.1 Motivational Factors in ODM Design

The main motivation for a hierarchical scheme to represent distance maps was to lower the memory requirements; however, we also wished to have reasonable performance in collision detection. Thus, we decided to augment the octree structure with precomputed distance indicators in order to limit the amount of searching which would otherwise be necessary with simply a standard octree. The distance indicators, however, must not add significantly to the memory requirement.

A fundamental question is what would the distance indicator(s) represent, and how they would be computed. *A problem arises when we attempt to associate a distance-to-nearest-obstacle to an octree node which is not at the lowest level of the tree (voxel level): the closest obstacle to that node may be different depending on from which location within that node the measurement is made.*

An illustration of the problem in the collision detection context can be seen in Figure 7.1. First, let us assume that we shall store only a single distance indicator within each *white* node W: the Manhattan distance to the closest *target node*, that is, a *grey* or *black* node *at the same level* L as W, measured in units of nodes of level L, rather than voxels. (We call this indicator the *nodal separation index* or NSI.) In Figure 7.1, the Manhattan distance (measured in level-L nodes) from W to target node T_1 is 1, while the Manhattan distance to T_2 is 2, and thus the NSI of W would be 1. In other words, we would be assuming that T_1 is "closer" than T_2. Let's say that our collision-detection scheme is such that only nodes with Manhattan distance from W equal to 1 would be checked for collision with obstacles. Then T_2 would not be examined and the collision with the obstacle area within T_2 (for the given robot position and radius r) would not be detected. The detection algorithm would therefore return a wrong result since no collision would be found with T_1.

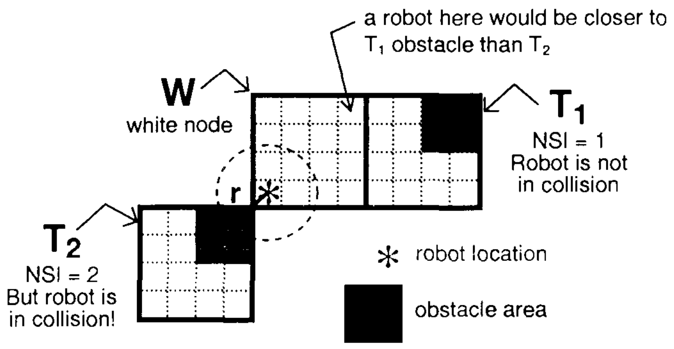

a robot here would be closer to T₁ obstacle than T₂
W
white node
T₁
NSI = 1
Robot is not in collision
r
T₂
NSI = 2
But robot is in collision!
robot location
obstacle area

Figure 7.1: A fundamental problem in hierarchical distance representation.

One simple way to avoid this problem is to use the NSI as a minimum search parameter. Starting with a Manhattan distance of NSI, the collision detection scheme would search for obstacles at the current distance and check for collisions until a collision were detected or until a maximum limit were reached. The maximum limit would be based on the node size of W and the robot radius length. Unfortunately, putting the the complete onus of finding the maximum search parameter on the collision detection algorithm (rather than on the algorithm to create the distance map structure) would severely affect the performance during collision detection. Therefore, we calculate minimum and maximum distance bounds during the creation phase and store a minimum and a maximum NSI for each *white* node within the distance map structure; we discuss these calculations in the next section.

7.2 Creation of Octree Distance Maps

The proposed *octree distance map* (ODM) is represented in a data structure similar to that of a conventional octree. The ODM node record is identical to an octree node record except that, for *white* child nodes which would otherwise (in a conventional octree) be represented as null pointers inside their parent *grey* nodes, the parent *grey* node record also contains two numbers indicating the minimum and maximum search parameters (NSI) for the *white* subnode; the maximum and minimum NSI are used during collision detection.

The computation of the minimum and maximum NSI's occurs during the ODM creation phase of our algorithm. Essentially, the algorithm searches outward from the *white* node W, at incrementally-increasing nodal distance, for obstacle nodes at the same level L as W. Two Euclidean distance measures, called *near-distance* and *far-distance*, are then computed between W and the obstacle node O (see Figure 7.2). First, assume that the obstacle node is ***black***. The near-distance is the distance between the respective regions represented by the nodes W and O, i.e., $d_E(W, O) = min_{w,o}\{d_E(w, o), w \in W, o \in O\}$. Let $\{PO\}$ indicate the set of points in O that correspond to the near-distance. To define far-distance, imagine that, for each point w in W, the distance to any point o in O were calculated. The far-distance is the maximum of these distances, i.e., $max_w\{d_E(w, PO), w \in W\}$. In the collision detection context, the near-distance is the upper bound on the robot radius such that, no matter where in W the robot is located, the robot cannot be in collision with the obstacle node O. *If the robot radius is smaller than the near-distance, it is guaranteed not to be in collision with the obstacle node.* The far-distance is a lower bound on the radius such that, no matter where in W the robot is located,it is certain to be in collision with O. *If the robot radius is greater than the far-distance, then the robot is guaranteed to be in collision with the obstacle node.*

Obviously, more than one pair of points in W and O may give the same near-distance or far-distance calculation result, but the distance values themselves are unique. Furthermore, since the nodes in the octree have a simple geometric shape (a cube), the near- and far-distances are easily computed using bounding boxes surrounding W and O, as shown in Figure 7.2. Also, note that if the obstacle node is *grey*, the near-distance and far-distance for that obstacle node are the minimums of the near-distance and far-distance calculations for each *black* subnode of the *grey* level-L node.

The near-distance and far-distance calculations are used to determine if an obstacle node is *ignorable*. An obstacle node O at level L is considered ignorable if the radius

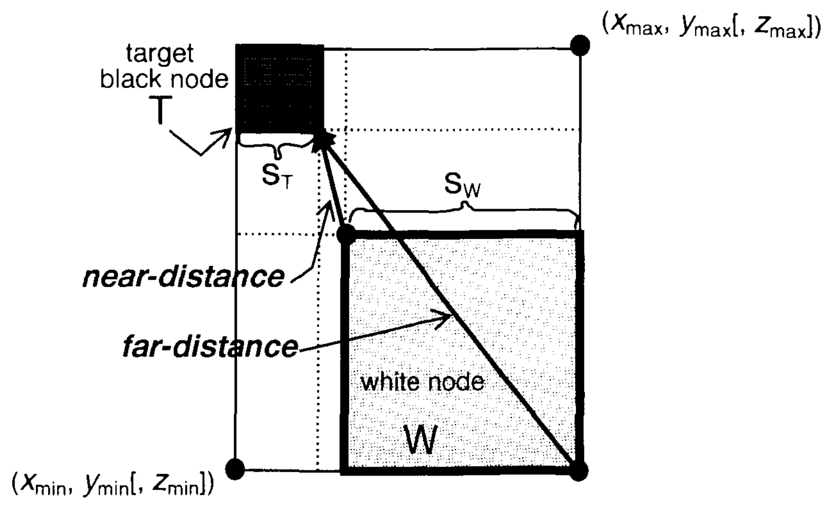
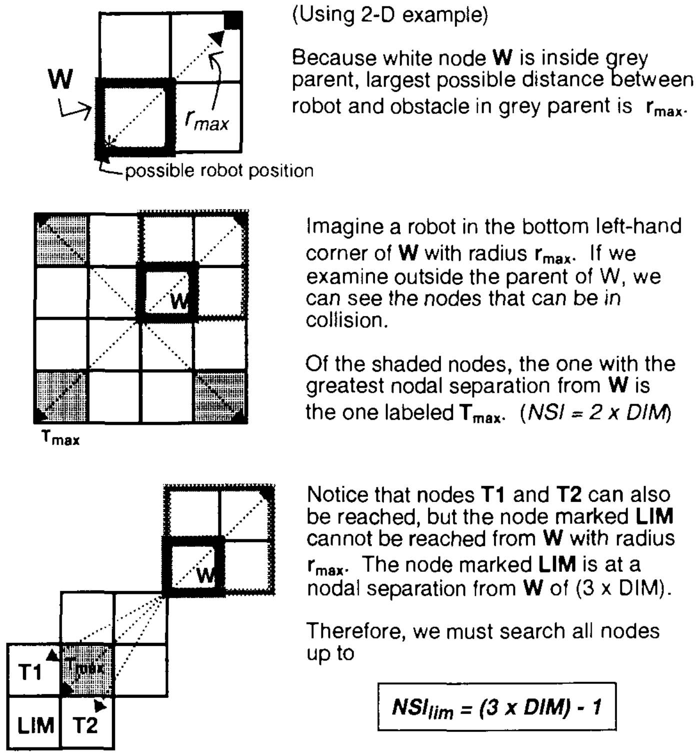


Figure 7.2: The concepts of near-distance and far-distance.

required for a robot in W to collide with O (i.e., the near-distance) is larger than the radius required for certain collision with some previous obstacle (i.e., the minimum far-distance). In other words, if this situation exists, a robot in W colliding with O will also undoubtedly collide with an obstacle that was previously found. Ignorable nodes are determined in order to lower the maximum-NSI bound and thus reduce time and computation requirements during collision detection. The maximum NSI for W is the largest Manhattan distance (units being nodes at the same level L as W) for which there are non-ignorable obstacle nodes.

As illustrated in Figure 7.3, the absolute upper limit on NSI at which a robot in W may be in collision is $(3 \times DIM - 1)$. The rationale for this limit is as follows: Node W is within a *grey* parent node; thus one of W's sibling nodes must be *grey* or *black*. The largest possible distance between a robot in W and some obstacle voxel in a sibling of W is the diagonal length of the parent of W. A robot in W with this distance as its radius can may be in collision with nodes of nodal separation up to, but not including, $(3 \times DIM)$; thus, $(3 \times DIM - 1)$ is the limit of our search for the maximum NSI. We could, of course, perform the search during collision detection (having stored a minimum NSI during ODM creation), proceeding outwards from the minimum NSI for W. However, by conducting the search during ODM creation to obtain both minimum and maximum NSI's, we can obtain better performance during collision detection.

As for the memory requirements for storing the minimum and maximum NSI,



(Using 2-D example)

Because white node **W** is inside grey parent, largest possible distance between robot and obstacle in grey parent is r_{max}.

Imagine a robot in the bottom left-hand corner of **W** with radius r_{max}. If we examine outside the parent of W, we can see the nodes that can be in collision.

Of the shaded nodes, the one with the greatest nodal separation from **W** is the one labeled T_{max}. (*NSI = 2 x DIM*)

Notice that nodes **T1** and **T2** can also be reached, but the node marked **LIM** cannot be reached from **W** with radius r_{max}. The node marked **LIM** is at a nodal separation from **W** of (3 x DIM).

Therefore, we must search all nodes up to

$\boxed{NSI_{lim} = (3 \times DIM) - 1}$

Figure 7.3: Illustration of maximum bound for maximum-NSI search.

because NSI's are calculated for *white* child nodes of *grey* nodes only, minimum NSI's will fall within the interval $[1, 3]$, requiring only 2 bits per NSI (in the 3-D case; in the 2-D case, minimum NSI will fall within the interval $[0, 2]$). (The maximum NSI will fall within the interval $[1, 8]$ and thus will require 3 bits per NSI.)

To demonstrate the construction of a 2-D ODM structure, we present the following example based on the binary image and quadtree in Figure 6.1. Our algorithm would operate as follows (please refer to Figures 6.1 and 7.4):

1. We start at the root of the tree and visit its four children in order. The leftmost child (child 0) is *grey*; we let this child be our current node and examine its four children. (Figure 6.1)

2. We are now at level 2 of the tree. Let child 0 be *CurrNode*. *CurrNode* is *white*; therefore, we must find its minimum and maximum NSI. (Figure 6.1)

3. We search level-2 nodes starting at a nodal separation of 1 and continuing to

the NSI limit of $(3 \times DIM - 1) = 5$. There are no *grey* or *black* nodes at NSI = 1, so we increment and search at NSI = 2. (Figure 6.1)

4. There are two level-2 target nodes at NSI = 2; thus the minimum NSI for $CurrNode$ is 2. We shall examine the node shown in Figure 7.4(a) first. Let this node be $Target$.

5. We recursively locate obstacle subnodes within $Target$ and determine that the near-distance is 10.0 and the far-distance is $\sqrt{349} = 18.68$. See Figure 7.4(b).

6. The next target node is shown in Figure 7.4(c). The near-distance is 4.0 and far-distance is 14.42. This now becomes $smallest\_far\_dist$.

7. We now let NSI = 3. There is one target node (shown in Figure 7.4(d)). Near-distance is 8.0, which is not less than $smallest\_far\_dist = 14.42$, and so this node is not ignorable. Far-distance is 17.89, so $smallest\_far\_dist$ is unchanged. Since there is a non-ignorable target node at NSI = 3, $temp\_max\_NSI$ is set to 3.

8. We now let NSI = 4. Here, there are two target nodes (Figure 7.4(e)). Target 1 has a near-distance of 16.12, which is greater than $smallest\_far\_dist = 14.42$, so this target node is ignorable. The far-distance is 26.0, so $smallest\_far\_dist$ is unchanged.

9. Target 2 at NSI = 4 is evaluated. Near-distance is $12.80 < 14.42$, so this target is not ignorable. The far-distance is 24.08, so $smallest\_far\_dist$ is unchanged. A non-ignorable target node has been found at NSI = 4, so $temp\_max\_NSI$ is set to 4.

10. NSI is now 5. There is one target node (Figure 7.4(f)). The near-distance is $18.86 > 14.42$, so the target is ignorable. No non-ignorable target nodes have been found at NSI = 5, so $temp\_max\_NSI$ remains unchanged.

11. We have reached the NSI limit for searching. We store $temp\_max\_NSI = 4$ as the maximum NSI for $CurrNode$.

12. This algorithm is repeated for all *white* nodes in the tree. The completed data structure is shown in Figure 7.5

We now state a high-level version of our ODM creation algorithm, **Build_ODM**, which generates an ODM given an octree for a non-empty workspace. (For a more detailed algorithm, please see Appendix B.)

```
Main routine: Build_ODM
Input: Octree representing voxel map.
Output: ODM representing hierarchical distance map.
  1. Augment octree nodes with minimum- and maximum-NSI storage fields
  2. For each white subnode of a grey node in the tree:
      (a) For all nodal distances $NSI$ = 1 to ($3DIM-1$), where DIM is the number of dimensions:
          i. Locate all target nodes at nodal distance $NSI$
          ii. For each black or grey target node:
              A. Calculate near-distance and far-distance for target
              B. If minimum NSI not stored, set minimum NSI = $NSI$
              C. Else:
                  • If near-distance > smallest far-distance for current white node, target is ignorable
                  • Else, target is non-ignorable
              D. Evaluate next target node (go to 2(a)ii)
          iii. If non-ignorable targets found at current nodal distance, set maximum NSI = $NSI$
          iv. Increment $NSI$ and go to 2a
      (b) Store min. and max. NSI for current white node
      (c) Find next white node (go to 2)
```

Although the creation algorithm is search-intensive, finding the maximum/minimum NSI parameters allows us to reduce the search-time during collision detection. Building the ODM is a one-time off-line preprocessing step.

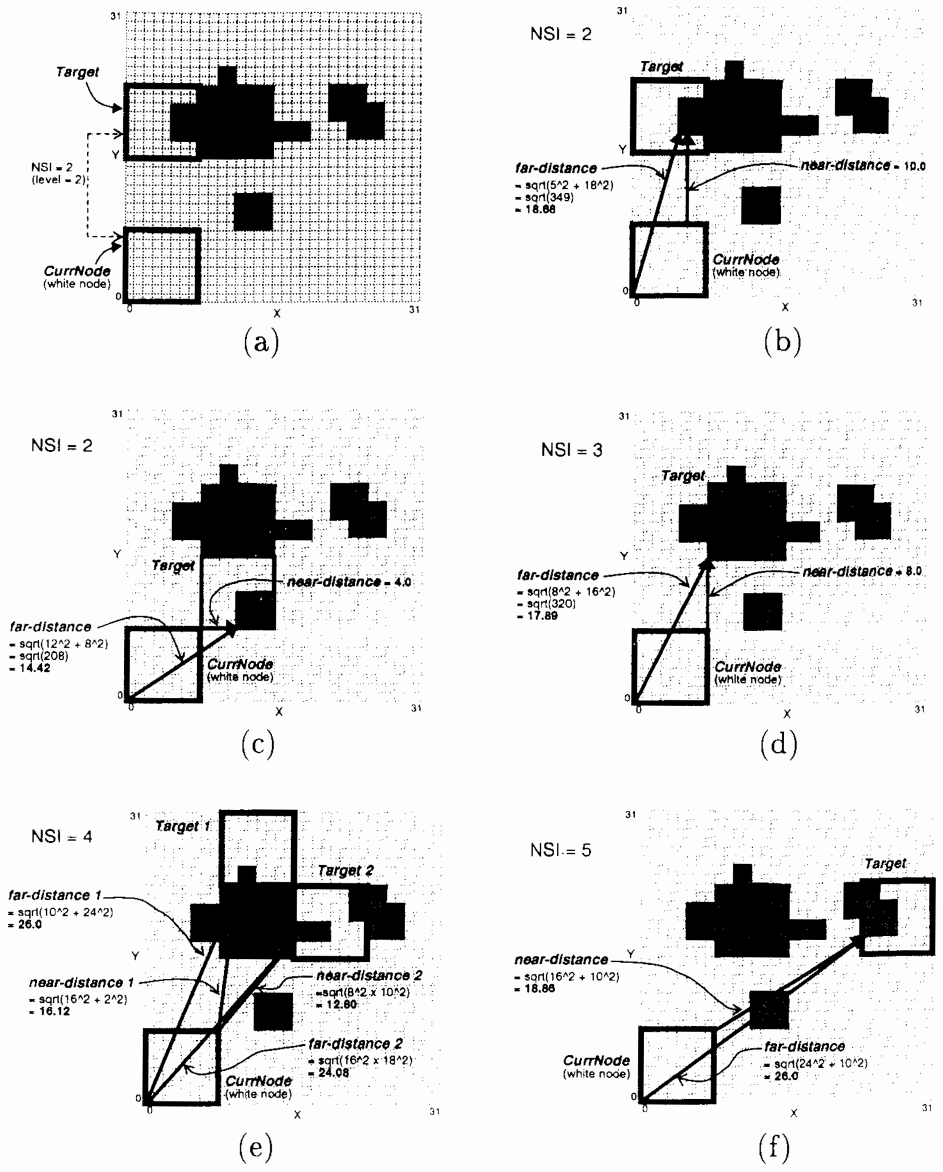

Target
NSI = 2 (level = 2)
CurrNode (white node)
(a)
NSI = 2
Target
far-distance = sqrt(5^2 + 18^2) = sqrt(349) = 18.68
near-distance = 10.0
CurrNode (white node)
(b)
NSI = 2
Target
near-distance = 4.0
far-distance = sqrt(12^2 + 8^2) = sqrt(208) = 14.42
CurrNode (white node)
(c)
NSI = 3
Target
far-distance = sqrt(8^2 + 16^2) = sqrt(320) = 17.89
near-distance = 8.0
CurrNode (white node)
(d)
NSI = 4
Target 1
Target 2
far-distance 1 = sqrt(10^2 + 24^2) = 26.0
near-distance 1 = sqrt(16^2 + 2^2) = 16.12
near-distance 2 =sqrt(8^2 x 10^2) = 12.80
far-distance 2 = sqrt(16^2 x 18^2) = 24.08
CurrNode (white node)
(e)
NSI = 5
Target
near-distance = sqrt(16^2 + 10^2) = 18.86
far-distance = sqrt(24^2 + 10^2) = 26.0
CurrNode (white node)
(f)

Figure 7.4: Illustrations for the ODM creation example.

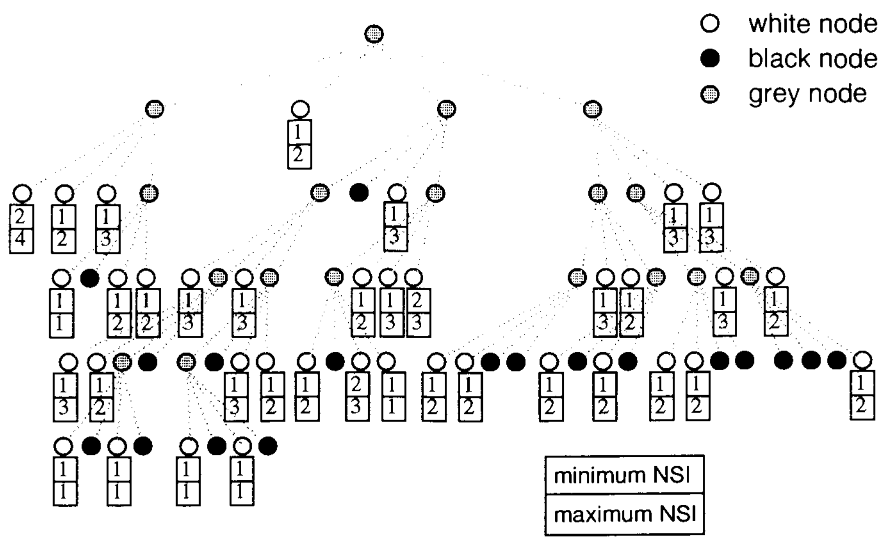


Figure 7.5: The octree distance map structure constructed for Figure 6.1; records under white nodes contain maximum and minimum NSI.

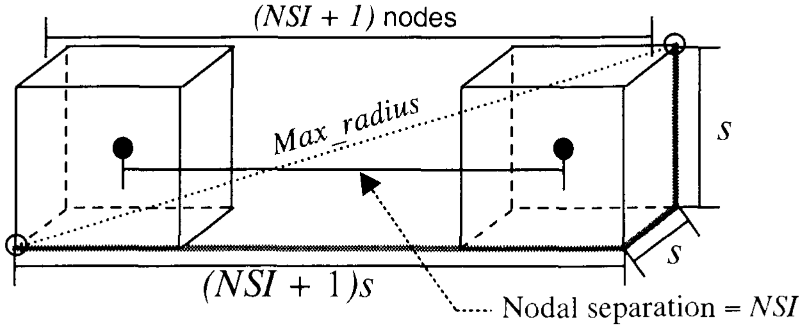
7.3 Collision Detection Using an ODM

After the octree distance map structure (the ODM) has been generated, it may be used in efficient collision detection. Here, let us assume that the robot is modeled using a number of spheres, and use a single sphere for the robot as an illustration. Note that a robot can be approximated to an arbitrary level of accuracy using spheres (del Pobil and Serna 1995).

Given a robot's center voxel and Euclidean radius, the algorithm below is used for collision detection. Note that robot radius may change from query to query without affecting the pre-generated ODM structure; the ODM is repeatedly used for each query without any further changes.

The algorithm first finds the *white* node W containing the robot in the octree. Then, with NSI equal to the min_NSI of W, the algorithm considers the set of *black* and *grey* nodes (target nodes) with nodal separation from W equal to NSI. (The target nodes are determined using the location-code process discussed in the **Background** section.) For each target node T, the algorithm calculates bounds on robot size (see below) to determine if the robot is or is not in collision with T, or if the collision situation is indeterminate. For the latter, the algorithm performs a recursive detection process: using the **Get_Distance** routine given in the previous section, the near-distance and far-distance (from W) to T are determined. If the robot size is less than the near-distance, no collision with T can occur. Otherwise, if T is *black*, the far-distance is compared with robot size. If the robot size is larger, a collision situation exists and a TRUE value is returned. If not, the algorithm calculates the near-distance between T and the robot center voxel (not W), and compares the distance with the robot size to evaluate the collision situation.

If T was *grey* and not *black*, the algorithm first creates a maximum-radius bound on the robot (see Figure 7.6). This bound implies that if the robot has a radius greater than the bound, the robot is in collision with *grey* node T no matter where the robot is located in the white node W. If such is the case, the algorithm immediately returns a TRUE collision result. Otherwise, the collision situation remains indeterminate. The algorithm proceeds to refine these bounds by further localizing the obstacle(s), identifying the *grey* and *black* child nodes of T. The child nodes are then evaluated using this recursive process until a TRUE/FALSE result is reached. If no collision is found with these target nodes, NSI is incremented (up to max_NSI for W) and the process is repeated until the bottom of the tree is reached and/or the collision situation is determined.



Largest radius for 2 nodes of side-length s, with nodal sep. NSI = length of diagonal of parallelepiped formed by string of (NSI + 1) nodes.

$$Max\_radius = \sqrt{[(NSI + 1)\, s]^2 + s^2 + s^2} = s \cdot \sqrt{(NSI + 1)^2 + 2}$$

Figure 7.6: Bound on maximum radius of robot; if robot is larger, a collision is detected.

We shall illustrate the use of ODM's in collision detection using the 2-D example from Figure 6.1. Suppose we define our robot with a center position of $(12, 6)$ (indicated by a star in the figure) and a radius of 6. The algorithm would operate as follows (the steps are illustrated in Figures 7.7(a) to (f).

1. Using the center position, the robot's location code is found to be 01320_4. We localize the robot to the *white* node at level 2 with reference point (8, 0). Let this node be W. (Figure 7.7(a))

2. We retrieve $min\_NSI = 1$ and $max\_NSI = 1$ from the parent of this node.

3. We find a single level-2 obstacle node at a nodal separation of 1 from the *white* node, and evaluate the collision situation. Let this node be T. (Figure 7.7(b))

4. Using the **Get_Distance** routine in Appendix B, we find the near-distance to be 0.0 (since the nodes are adjacent) and the far-distance to be 8.0. Since the far-distance is greater than the robot radius, the collision situation is indeterminate. (Figure 7.7(c))

5. Because T is *grey*, we first evaluate the maximum-radius condition. The maximum radius (the robot radius which would make collision a certainty) is $17.89 > 6.0$; the situation remains indeterminate. (Figure 7.7(d))

6. We localize the obstacle: T becomes child 1 of the previous T. The robot is localized to quadrant 3 of W (which becomes the new W), and we increment the level to 3. The recursive routine is called again with these new parameters. (Figure 7.7(e))

7. For the new T and W, we find the near-distance to be 0.0 and the far-distance to be 4.0. Because T is *black*, we can compare the robot size to the far-distance. Because the robot size is greater ($6.0 > 4.0$), we can conclude that there is a collision. The algorithm returns a TRUE result. (Figure 7.7(f))

We now state a high level version of the algorithm, **ODM_Detect**. (For a more detailed algorithm, please see Appendix C.)

Main routine: **ODM_Detect**
Input: octree distance map, robot position, robot radius.
Output: collision situation: TRUE if collision, else FALSE.

```
1. Localize robot to the highest white node in tree
2. Retrieve $min\_NSI$ and $max\_NSI$
3. For $NSI = min\_NSI$ to $max\_NSI$ do:
    (a) Generate list of black and grey target nodes with nodal distance equal to $NSI$
    (b) For each target node in list
        i. Call Recurse_Detect to obtain collision result
        ii. If there is a collision
            • Return TRUE result
        iii. Otherwise, go to next target node (go to 3b)
    (c) No collision found; increment $NSI$ and go to 3
4. No collision has been found. Return FALSE result.
```

Subroutine: **Recurse_Detect**
Input: robot *white* node W, position, radius, target node T *Output*: collision situation: TRUE if collision, else FALSE.

```
1. Calculate near- and far-distance for $W$ and $T$
2. If robot radius < near-distance
    • No collision; return FALSE
3. Else, if ($T$ is black)
    • If robot radius > far-distance
        – Collision is certain; return TRUE
    • Else:
        – Calculate shortest distance between robot position and $T$ (i.e., near-distance with
          robot voxel as the white node)
        – If robot is bigger than shortest distance
            * Return TRUE collision result
        – Else, no collision; return FALSE result
4. Else (if $T$ is grey)
```

- Calculate the maximum-radius bound for W and T (see Figure 7.6)
- If robot is larger than bound /*certain collision*/
  - Return TRUE result
- Else, recursively call **Recurse_Detect** with all *black* and *grey* children of T as target nodes
- If any result from the recursive call is TRUE
  - Return TRUE result
- Else, if no TRUE result after all children evaluated
  - Return FALSE result /*No collision with T*/

For a robot modeled as multiple spheres, the following algorithm is then used:

Main routine: **ODM_Robot_Detect**
Input: ODM, number of robots *N_robots*, robot centers and radii.
Output: collision situation: TRUE if collision, else FALSE.

1. For *Index* = 1 to *N_robots* do:
   (a) Call **ODM_Detect**, passing center location and radius for robot[*Index*]. Get collision result in *Result*.
   (b) If *Result* is TRUE, output TRUE and end.

Note that while the collision detection algorithm is specifically written for use with spherical robot models, it may also be extended for use for collision detection with line segments (see Appendix D).

Another extension of our work is that of distance estimation, i.e., determining the distance from the robot to the nearest obstacle. However, the algorithm would need to be modified such that the minimum distance to obstacle must be kept and updated whenever a *black* node were evaluated to be closer than any previous one.

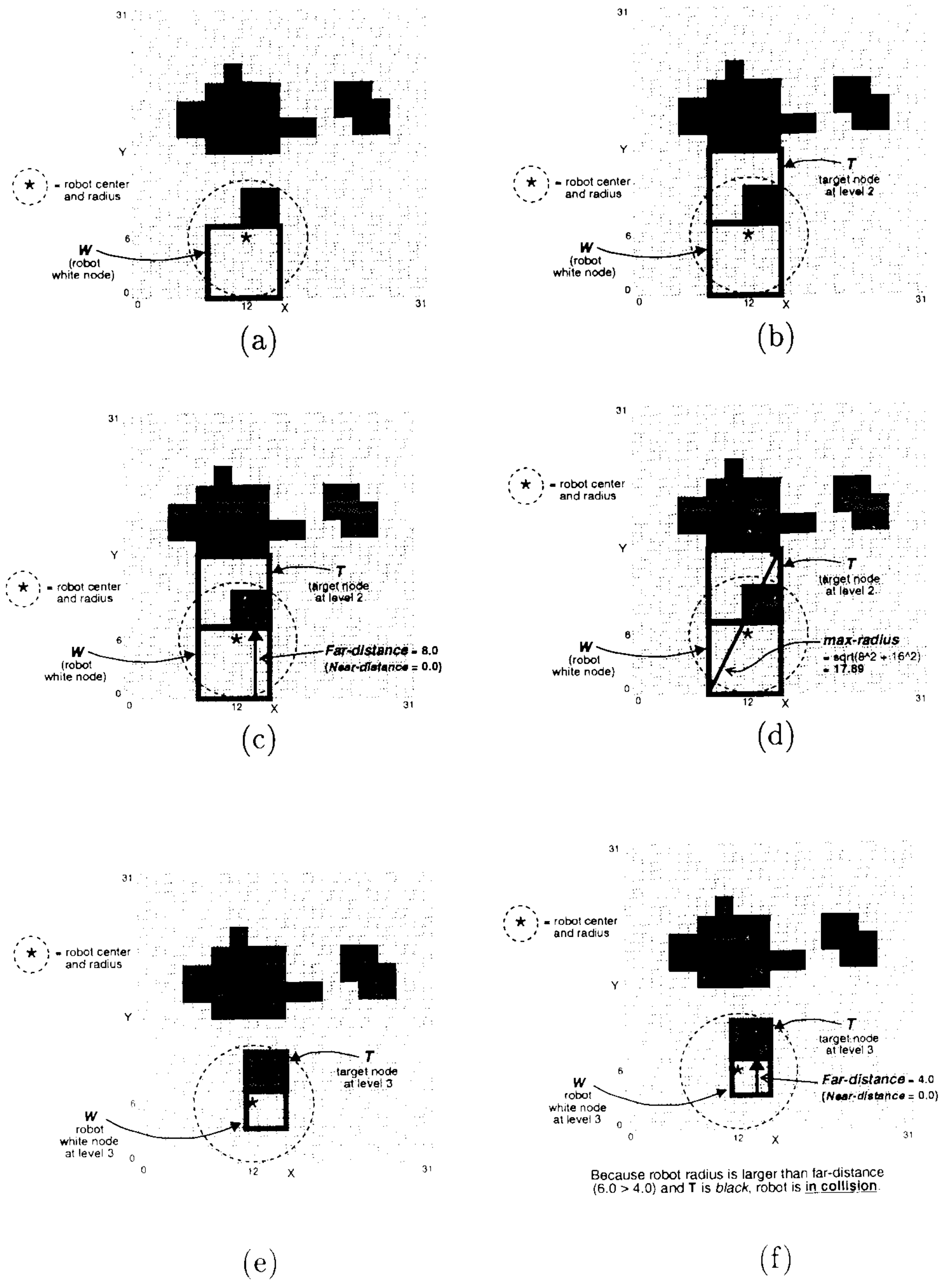

= robot center and radius
W (robot white node)
T target node at level 2
Far-distance = 8.0 (Near-distance = 0.0)
max-radius = sqrt(8^2 + 16^2) = 17.89
target node at level 3
W robot white node at level 3
Far-distance = 4.0 (Near-distance = 0.0)
Because robot radius is larger than far-distance (6.0 > 4.0) and T is black, robot is in collision
(a)
(b)
(c)
(d)
(e)
(f)

Figure 7.7: Illustrations for the ODM collision detection example.

Chapter 8

Experiments

Both the **ODM_Build** and **ODM_Detect** algorithms were implemented in C on a Sun Sparcstation platform, and several experiments were performed. In the first experiment, the same 32 × 32 quadtree in Figure 6.1 was input into the creation algorithm and generated the correct distance map output. I executed the detection algorithm for every pixel in the grid for robots with Euclidean radii of 1, 2, and 3 units. These results are shown in Figure 8.1. As illustrated, the algorithms work.

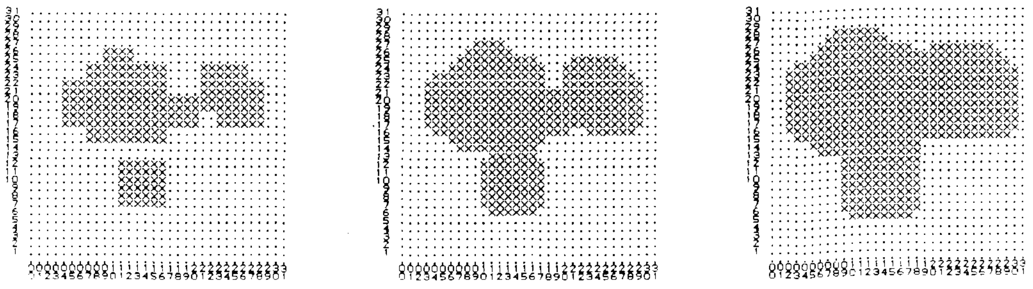

(a) (b) (c)

Figure 8.1: Results of 2-D experiments with data from Figure 2 for robot radii of 1, 2, and 3 voxels (a, b, and c respectively). (x indicates collision).

Next, I performed experiments on 3-d examples. I ran the algorithms on voxel maps of five different 3-D workspaces, each discretized in a 100 × 100 × 100-voxel array. Table 8.1 lists the memory requirements of each ODM, along with the requirements for each unaugmented octree and the constant memory requirement of a voxel-array-based Euclidean distance map. From the results, we can see that the the ODM requires about 25 percent more memory compared to the amount required for the corresponding unaugmented octree; the extra memory is the result of the five-byte requirement for storage of minimum and maximum NSI. However, the ODM's memory requirements are significantly less than those of a voxel-based distance map: the ODM uses between 70 to 95 percent less memory.

The next experiment involved measuring the speed performance using both ODM's and unaugmented octrees in 3-d collision detection. The algorithm used for octree collision detection was simply the **ODM_Detect** algorithm without the benefit of stored minimum and maximum NSI; for the unaugmented octree, the minimum NSI is assumed to be 1 and maximum NSI is assumed to be $(3 \times DIM) - 1 = 8$. Thus, the octree collision detection algorithm, not being a dedicated octree algorithm, may not be the most efficient one available, and other, more dedicated octree collision detection algorithms may yield better performance.

In comparing ODM versus octree performance, both a count of condition checks (when a comparison is made between the robot size and a number) and the average time required on a Sparc-10 for each collision check (in milliseconds) were measured. The experiment consisted of 3 sets of 1000 collision detections each for the five workspaces, using random robot positions and radius sizes (each set having a different range of radii). The results are shown in Tables 8.2 and 8.3. There is a substantial improvement in performance using ODM's over octrees, both in the number of condition checks (ODM: 10 to 48% (35% on average) fewer) and in run-time (ODM: 20 to 50% (39% on average) faster). These results stand to reason: a great deal of the searching done when using an unaugmented octree in collision detection is performed when an ODM is created.

The third series of tests involved performing collision detection for complete robot models each comprising 159 spheres. Figure 8.2 shows an example of a robot modeled by spheres. I tested four separate robot configurations in the same workspace – two configurations not in collision, two in collision. Again, the performance of ODM collision detection is compared with collision detection using an unaugmented octree; the results are shown in Table 8.4. We once again see a drastic improvement in performance (between 28 to 58%, average 43%) when using ODM's for collision detection.

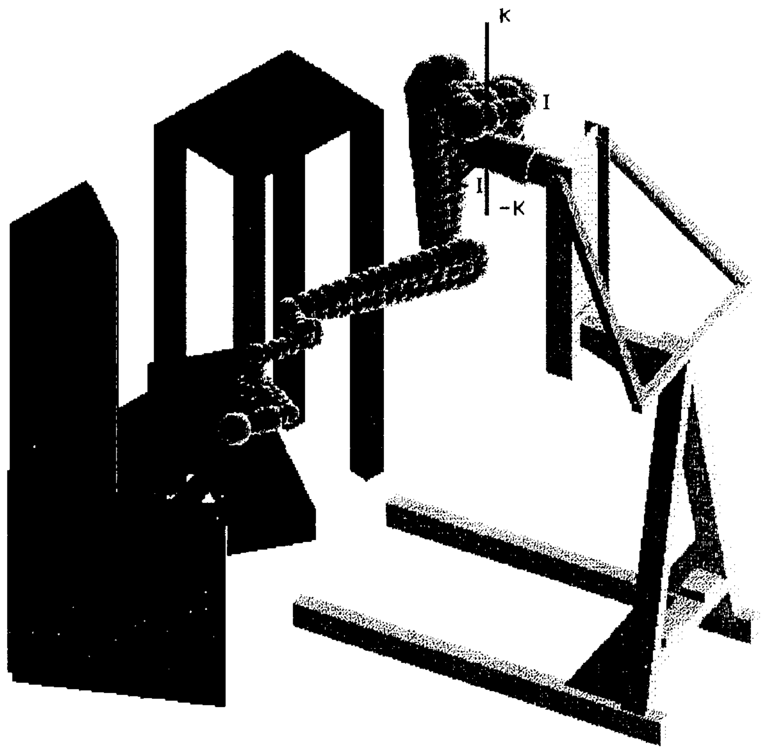


Figure 8.2: Example of a workspace used for collision detection tests; robot spherical model shown.

Workspace	Memory usage		% Extra
	ODM	Octree	memory
1	594,256	468,552	27
2	653,276	516,720	26
3	1,169,832	919,352	27
4	234,240	186,536	26
5	1,018,776	804,376	27
Voxel-based dist. map (100^3 array, 4 bytes / voxel			4,000,000

Table 8.1: Memory usage for ODM, octree, and voxel-based distance map for five workspaces.

		Avg. # of condition checks / test				
Max	Data	Workspace				
size	Structure	1	2	3	4	5
10	Octree	26.0	74.1	62.5	75.9	73.2
voxels	ODM	13.7	38.3	34.1	58.4	38.2
% Improvement		47	48	45	23	48
25	Octree	12.3	40.9	34.5	62.3	41.3
voxels	ODM	6.7	25.8	22.2	53.6	26.4
% Improvement		46	37	36	14	36
40	Octree	9.7	23.4	24.1	53.2	22.7
voxels	ODM	5.3	14.2	15.3	48.2	14.0
% Improvement		45	39	36	9	38

Table 8.2: Condition-check results of 3-D collision detection using ODM and octree. (1000 tests per avg.)

		Avg. time per test (ms)				
Max	Data	Workspace				
size	Structure	1	2	3	4	5
10	Octree	4.5	13.4	11.2	16.6	13.4
voxels	ODM	2.3	6.6	5.8	11.1	6.7
% Improvement		48	51	48	33	50
25	Octree	2.1	7.2	6.0	12.7	7.2
voxels	ODM	1.2	4.3	3.7	9.5	4.3
% Improvement		45	40	39	25	40
40	Octree	1.8	4.2	4.2	10.0	4.1
voxels	ODM	0.9	2.5	2.5	8.0	2.4
% Improvement		47	41	39	20	41

Table 8.3: Times (Sparc-10) for 3-D collision detection, using ODM and octree. (1000 tests per avg.)

	Robot configuration			
	1	2	3	4
# Spheres in collision	none	none	10	19
Octree: Total time required	5245	3841	5157	4877
Octree: Avg time/sphere	33.0	24.2	32.4	30.7
ODM: Total time required	3254	2765	2799	2063
ODM: Avg time/sphere	20.5	17.4	17.6	19.3
% Improvement	38	28	46	58

Table 8.4: Speed performance (Sparc-10) for 3-D collision detection with 4 robot configurations (159 spheres) using ODM and octree. (Times in ms.)

Part III

Conclusions

Chapter 9

Conclusion and Future Work

This thesis explored the use of real-world, sensor-based data for modeling three-dimensional workspaces and for performing collision detection and path planning for a robot within that workspace. Specifically, the thesis examined ways to integrate range images to model a 3-d workspace. In addition, the thesis introduced a way of converting such a model to a form that is conducive to efficient collision detection and path planning, and algorithms for the creation and use of such a model are presented.

For multiple-view range image integration, we needed to look at the intended use of the generated 3-d model of the workspace. Since the model is to be used for robot path planning and collision detection, the model should have several characteristics: (i) a spatial occupancy model using voxels, which is quicker to generate and less complex than modeling real-world scanned data with CAD primitives; (ii) obstacles represented as solid groups of voxels rather than as shells enclosing empty space to avoid path planning problems where, for instance, the robot's starting position is put inside an object; and (iii) unscanned areas to be represented as obstacle areas, so that the path planner avoids moving the robot into unknown regions of the workspace.

The first step for integrating range images was to determine how to geometrically relate the individual range images, taken from several points of view, into a single frame of reference. Because the laser scanner was mounted on a Puma 560 robot, the robot yielded transformations for each scanner position. By fixing the location of the range image reference frame and then relating it to the PUMA global frame, it became possible to relate range images from multiple views to the PUMA frame.

Two approaches were examined for integrating range images. The first, direct mapping, maps range images into workspace voxels on a one-to-one basis. This method produced shell representations of objects and did not take into account the unscanned workspace areas, so was deemed unfit for path planning purposes, but this method did produce shell representations which verified the correctness of the transforms.

The second approach, peeling, uses the mirror and camera positions for each range image pixel to determine in which directions to remove voxels from a complete "block" of voxels. The approach uses the concept that points which have been scanned by the laser scanner must have direct light paths to both the laser source (mirror) and imager (camera), and therefore these paths must be free of voxels. The advantages of this approach are that 1) since voxels are removed away from the surfaces of objects but not below the surface, the leftover voxels form solid representations of the objects; and 2) unscanned space does not have voxels removed, and so are treated as obstacles.

Based on the results of integration via peeling, we see that the shapes of objects have begun to be uncovered. However, due to the limited number of scans and the limited scanned area of each scan, not enough voxels are removed from the block to provide us with clear pictures of the objects for this thesis. When viewed in 3-D animation, shapes are more easily discernable in the integrated voxel map. The results showed the previously mentioned advantages of the peeling algorithm, namely solid representations of objects and voxelized representation of unscanned areas.

A generated voxel map can be made much more efficient for collision detection by converting it into a distance map. Large voxel-based distance maps, however, use impractical amounts of memory. In the second part of this thesis, I presented a novel hierarchical representation for distance maps, called *octree distance map*, or ODM, which utilizes the memory efficiency of octrees for the purposes of collision detection and robot path planning, without a large sacrifice in terms of performance. The ODM is based on the conventional octree data structure, but is augmented with *nodal separation indices* or NSI, which provide distance-to- closest-obstacle information for each *white* node in the octree while keeping memory requirements low. This thesis presented algorithms for creating an ODM from a conventional octree, and for utilizing the ODM in collision detection. The experiments using these algorithms indicated that the use of the ODM (i) is correct, (ii) provides a drastic reduction in memory requirements over voxel-based distance maps, and (iii) exhibits a significant improvement in collision-detection performance, both in condition-check comparisons and in overall speed, over the use of unaugmented octrees for collision detection, at the cost of slightly higher memory requirements. On average, ODM collision detection is 35 to 40 percent faster and more efficient than when performing the same tasks using an unaugmented octree. In future, we intend to run further experiments based on our approach, and explore different applications for the ODM. We will also investigate possible memory efficiency improvements, as well as certain theoretical aspects; for instance, rather than storing only the minimum and maximum NSI, what distance information can be stored to further improve collision detection efficiency without seriously affecting memory efficiency.

9.1 Future Work

One direction of future work that holds promise is the exploration of the memory/performance tradeoff of the ODM. Specifically, it may be possible to include more distance information within the data structure – thus using more memory, though still much less than a voxel-based distance map – in order to gain better collision detection performance. One feasible approach is to store some perimeter distance function for each white node, rather than a single distance indicator. Collision detection would involve determining the closest point of the node perimeter to the robot sphere center point, evaluating the perimeter distance function for that perimeter point, and comparing the result to the robot sphere radius. Such an approach would avoid the recursion required for ODM collision detection, thus improving performance.

References

Ahuja, N. and J. Veenstra (1989, February). Generating Octrees from Object Silhouettes in Orthographic Views. *IEEE Transactions on Pattern Analysis and Machine Intelligence 11*(2), 137–149.

Arimoto, S., H. Noborio, S. Fukuda, and A. Noda (1988). A Feasible Approach to Automatic Planning of Collision-Free Robot Motions. In B. R. R. Bolles (Ed.), *International Symposium on Robotics Research*, pp. 479–488. MIT Press.

Asada, M. (1990, November-December). Map Building for a Mobile Robot From Sensory Data. *IEEE Transactions on Systems, Man and Cybernetics 20*(6), 1326–1336.

Bandi, S. and D. Thalmann (1995, September). An Adaptive Spatial Subdivision of the Object Space for Fast Collision Detection of Animated Rigid Bodies. *Computer Graphics Forum 14*(3), C/259–C/270.

del Pobil, A. P. and M. A. Serna (1995). *Spatial Representation in Motion Planning.* Number 1014 in Lecture Notes in Computing Science. Springer.

del Pobil, A. P., M. A. Serna, and J. Llovet (1992). A New Representation for Collision Avoidance and Detection. In *1992 IEEE International Conference on Robotics and Automation.*

Egbert, P. K. and S. H. Winkler (1996, July). Collision-free object movement using vector fields. *IEEE Computer Graphics and Applications 16*(4), 18–24.

Gargantini, I. (1982, December). Linear Octtrees for Fast Processing of Three-Dimensional Objects. *Computer Graphics and Image Processing 20*(4).

Grandjean, P. et al. (1989). 3-D Modeling of Indoor Scenes by Fusion of Noisy Range and Stereo Data. In *Proceedings. 1989 IEEE International Conference on Robotics and Automation*, Volume 2.

Greenspan, M. and N. Burtnyk (1996). Obstacle Count Independent Real-Time Collision Avoidance. In *Proceedings. 1996 IEEE International Conference on Robotics and Automation.*

Gupta, K. K. and X. Zhu (1995). Practical global motion planning for many degrees of freedom: A novel approach within sequential framework. *Journal of Robotic Systems 12*(2), 105–118.

Hayward, V. (1986). Fast Collision Detection Scheme By Recursive Decomposition of A Manipulator Workspace. In *1986 IEEE International Conference on Robotics and Automation*, Volume 2, pp. 1044–1049.

Herman, M. (1985). Generating Detailed Scene Descriptions from Range Images. In *IEEE International Conference on Robotics and Automation.*

Jarvis, R. (1993). Distance Transform Based Path Planning for Robot Navigation. *Recent Trends in Mobile Robotics* .

Jung, D. and K. Gupta (1997). Octree-based hierarchical distance maps for collision detection. *to appear in Journal of Robotic Systems* , 26 manuscript pages.

Latombe, J. C. (1991). *Robot Motion Planning.* Kluwer Academic Publications.

Leonard, J. J. and B. A. Moran (1992). Sonar Data Fusion for 3-D Scene Reconstruction. In *Proceedings of the SPIE - The International Society for Optical Engineering*, Volume 1828.

Lin, X. and W. G. Wee (1985). Shape Detection Using Range Data. Published by IEEE.

Martin, W. N. and J. K. Aggarwal (1983, March). Volumetric Descriptions of Objects from Multiple Views. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-5*(2), 150–158.

Noborio, H., S. Fukuda, and S. Arimoto (1988, November). Construction of the Octree Approximating a Three-Dimensional Object by Using Multiple Views. *IEEE Transactions on Pattern Analysis and Machine Intelligence 10*(6), 769–781.

Noborio, H., Naniwa, and S. Arimoto (1990). A Quadtree-Based Path-Planning Algorithm for a Mobile Robot. *Journal of Robotic Systems 7*(4), 555–571.

Potmesil, M. (1987). Generating Octree Models of 3D Objects from Their Silhouettes in a Sequence of Images. *Computer Vision, Graphics, and Image Processing 40*, 1–29.

Roth-Tabak, Y. and R. Jain (1989, June). Building an Environment Model Using Depth Information. *Computer* .

Samet, H. (1990a). *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Don Mills, Ontario: Addison-Wesley Publishing Company.

Samet, H. (1990b). *The Design and Analysis of Spatial Data Structures.* Don Mills, Ontario: Addison-Wesley Publishing Company.

Sharma, A. and S. A. R. Scrivener (1990). Computer Vision Based Automatic Construction of 3-D Geometry Scene Models. In *UK IT 1990 Conference*, pp. 71–78. IEE.

Shneier, M. (1981). Path-length distances for quadtrees. *Information Sciences 23.*

Stenstrom, J. R. and C. I. Connolly (1986). Building Wire Frames from Multiple Range Views. Published by IEEE.

Stenstrom, J. R. and C. I. Connolly (1992). Constructing Object Models from Multiple Images. *International Journal of Computer Vision 9*(3).

Succi, G. et al. (1990). 3D Feature Extraction from Sequences of Range Data. In *Robotics Research. Fifth International Symposium.* MIT Press.

Xu, Z. (1992, June). Extracting Complete 3-Dimensional Boundary Representation from Multiple Range Images. Master's thesis, Simon Fraser University.

Yang, H., K. Gupta, and M. Greenspan. Path Planning with Distance Map Based Efficient Collision Detection. Technical report, Simon Fraser University, in preparation.

Yao, H., R. Podhorodeski, and D. Zuomin (1993). A Cross-Section Based Multiple-View Range Image Fusion Approach. In *Proceedings of the SPIE - The International Society for Optical Engineering*, Volume 1956.

Appendix A

Range Scanner System Calibration

A.1 Calibration Procedure

The laser range scanner was calibrated using the software and V-shaped calibration block (hereafter called the V-block) provided by Technical Arts. (The calibration block is shown in Figure 4.4.) The software performed the following required functions:

1. Oscillation of a mirror which produces a plane of laser light. The intersection of this plane with various surfaces is captured by the camera to calculate range.

2. Stepping of a second mirror, thus stepping the light plane across the workspace and producing a 2-D range image.

3. Determination of two camera orientation parameters, pitch and roll, based on the shape of the light plane intersecting the V-block.

The software also included an automatic calibration procedure which, assuming proper camera/laser alignment and accurate camera/laser parameters entered by the user, stepped the light plane through a user-defined range of positions and performed calibration at each X-Y point in its image space.

Thus, the steps to calibrate the laser scanner are:

1. Measurement of camera/laser parameters:

    angle of the light plane at zero position angle of the camera with respect to vertical

height of the V-block

vertical distance from top of V-block to laser

vertical distance from top of V-block to camera

2. Adjustment of the camera's orientation, based on the pitch and roll parameters calculated by the software. (Pitch and roll are required to be between -1.0 and 1.0 degrees.)

3. Execution of the automatic calibration procedure. This procedure is quite slow on a 386 PC, averaging approximately 10 lines or light plane positions per minute.

4. At the PUMA data terminal, type `show t6*tool`. This command will display a matrix, T6O, and transform angles O, A, and T (which correspond with ϕ, θ, and ψ in Euler Z-Y-Z rotation). Write down the first three elements of the fourth column of T6O (which are the (X, Y, Z) coordinates of the hand with respect to the PUMA) and the transform angles; these parameters provide the transform for the initial calibration position.

5. Finally, use a tape measure to measure the positions of the oscillating mirror and camera lens in terms of displacement from the PUMA tool center point (center of the PUMA tool flange) in the X, Y, and Z directions (relative to the tool frame).

To perform the PUMA-to-scanner calibration, the following steps are required:

1. Attach a measurement device (e.g. a plumb line) of known length L to the end-effector of the PUMA, such that the device hangs directly vertically. The displacement of the point of attachment A from the PUMA's tool reference point must also be measured; this displacement is expressed in reference to the tool reference frame position for calibration S_0 as ${}^{S_0}A$. With these two measurements known, the position of the measurement device with respect to S_0 is ${}^{S_0}D = {}^{S_0}A - [0, 0, L]$.

2. With the V-block in the position in which the laser scanner was calibrated, touch the tip of the measurement device to the corner of the V-block which is used as reference in range scanner calibration (see Figure A.1). Obtain the position of the tool reference point from the PUMA controller ($^{\mathbf{P}}_{\mathbf{S_0}}T$). Since we know the position of the measurement device with respect to $\mathbf{S_0}$ from the previous step, we can determine $^{\mathbf{P}}D = {}^{\mathbf{P}}_{\mathbf{S_0}}T\,{}^{\mathbf{S_0}}D$.

3. Since this corner of the calibration block is (0, 0, 0) in the scanner's image space, and since we know the position of the tip of the measurement device with respect to PUMA (from the previous step), we now have the coordinates in the PUMA frame of reference for the center of the reference frame $\mathbf{G_0}$ of the scanner's image space and thus the transform from PUMA space to scanner space, as discussed in Section 3.2.3.

4. Using the dimensions of the V-block, the measurement device tip may be touched to other corners of the V-block in order to verify the accuracy of the transform calculated above.

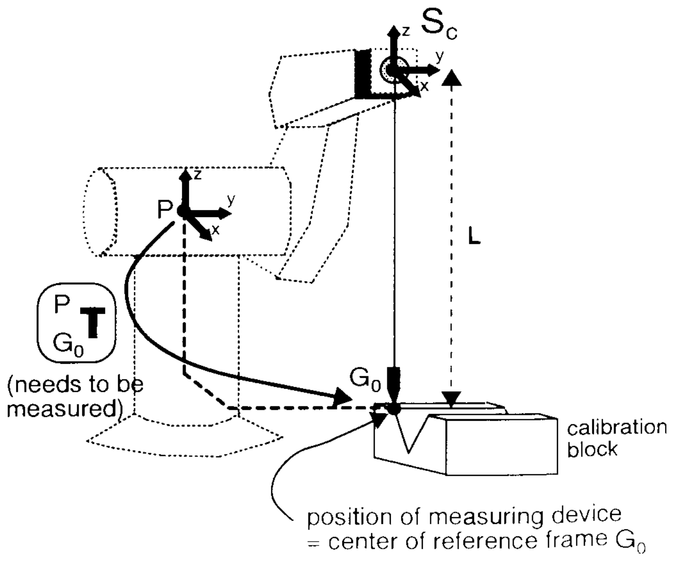


Figure A.1: PUMA-to-scanner calibration using the V-block.

A.1.1 Capture of Multiple-View Range Images

Once the scanner system is calibrated, the range images are captured using the following procedure:

1. Move the robot to a suitable position via teach pendant or robot control program. The selected position should allow the camera to image a significant part of the unscanned object space. Roughly keep track of which parts of the object space have been scanned by noting the general direction of the camera in relation to the object space.

2. Once the robot is in position, obtain the robot transform from the PUMA controller via the data terminal by typing `show t6*tool`. Write down (X, Y, Z) and (ϕ, θ, ψ) as before.

3. Type `show joint` and write down the joint angles J1-J6 in case the experiment must be repeated at a later date.

4. On the scanner PC, examine the current start and end positions of the scanner. If the scan area is too large for the portion of object space you are scanning, change the positions; this will decrease your scan time. Note the start and end positions.

5. Enter a scan file name. The range image will be written to this file.

6. Start the scan.

7. When the scan is complete, repeat steps 1-6 until the object space has been satisfactorily scanned.

A.2 Offline Integration of Range Images

(Note: This section assumes that the required Unix executables are in the Unix scan directory.) Once scanning has been completed, the range image files are integrated offline. The integration process is as follows:

1. Convert each of the image files from binary format to ASCII text by running the program `rdascii`. At the scanner PC, change the directory to `c:`
   `100at`
   `data` and type the following command for each image:

   ```
   rdascii > text-file-name.txt
   ```

   where `text-file-name.txt` is the name of the new text file. Then type the scan file name entered at the scanning setup screen. The resulting text file contains a list of the 3-d coordinates of all scanned points in the image.

2. Copy the text files to a 3.5" floppy disk and insert the disk a Sun workstation. On the workstation, go to your scan integration directory and copy the files to the Sun file system by typing:

   ```
   mcopy ''a:*'' .
   ```

3. For each text file, start an editor (e.g., EMACS) and manually insert the following information on the top line of the file:

   `X-coord Y-coord Z-coord` ϕ`-angle` θ`-angle` ψ`-angle`

   This information was obtained from the PUMA controller via the data terminal.

4. Create a file in the Unix scan directory called `HO.DATA`. In this file, manually enter on a single line the (X, Y, Z) and (ϕ, θ, ψ) coordinates of the initial calibration position of the PUMA:

   `X Y Z` ϕ θ ψ

5. Create a file in the same directory called `LASER.DATA`. On two separate lines, manually enter the (X, Y, Z) displacement of the laser oscillation mirror and camera lens:

   ```
   laser-X laser-Y laser-Z camera-X camera-Y camera-Z
   ```

6. Create a third file called `CAL.DATA`. In this file, manually enter the (X, Y, Z) displacement D_p' of the V-block origin point. Enter three zeroes following the displacement:

   ```
   X Y Z 0 0 0
   ```

7. Finally, type the command

   ```
   integ
   ```

   and enter the appropriate information when prompted (number of scans, file names, output voxel map file). The resulting voxel map file may be viewed on an SGI Indigo workstation or on a Sun workstation remotely connected to an SGI (see Section 4.1.4).

A.3 Display of integrated voxel map

Display of the voxel map is performed by a program on the SGI Indigo workstation using OpenGL routines. Once the voxel map file has been generated, follow this procedure to display the voxel map.

1. Copy the generated voxel map file to your viewing directory (if the viewing program does not exist in the scan directory).

2. Use the command `show3d <voxel-map> <d|c>` where `voxel-map` is the name of the generated voxel map file and `d|c` indicates whether to display voxels as dots or cubes.

3. Once processing is complete, a window displays the voxel map within a wire-frame cube denoting the extent of the voxel mapped space. Use the mouse pointer position to control the speed and direction of rotation of the display. Use the - and + keys to zoom in and out of the scene.

4. When you have finished viewing the voxel map, close the window to exit the viewing program.

Appendix B

Detailed ODM Creation Algorithm

Main routine: **Build_ODM**

Input: Octree representing voxel map.

Output: Octree Distance Map representing hierarchical distance map.

```
1. Augment octree nodes with minimum- and maximum-NSI storage fields
2. Beginning at root, traverse across all eight children and down tree. For each white node
   $CurrNode$ within a grey node
    (a) Let $L$ = level of $CurrNode$
    (b) Initialize $NSI\_limit$ to $(3 \times DIM - 1)$, where DIM is the dimension of space. (See
        Figure 7.3)
    (c) For $NSI\_Dist = 1$ to $NSI\_limit$ do
        i. Locate all nodes of level $L$ with nodal separation from $CurrNode$ equal to $NSI\_Dist$.
           Store nodes in array $Targets$
        ii. For each member $Targets[i]$
            A. If ($Targets[i]$ is black or grey)
                • Call Get_Distance, passing $CurrNode$, $Targets[i]$, white node level, tar-
                  get node level (both levels equal to $L$), and receiving $near\_dist$, $far\_dist$
                • If ($smallest\_far\_dist$ is not initialized or $far\_dist < smallest\_far\_dist$)
                  – Let $smallest\_far\_dist = far\_dist$
                • If ($CurrNode.min\_NSI$ has not been initialized)
                  – Let $CurrNode.min\_NSI = NSI\_Dist$
                • Else /*Determine if $Targets[i]$ is ignorable*/
                  – If ($near\_dist \geq smallest\_far\_dist$)
                        $Targets[i]$ is ignorable
            B. Evaluate next $Targets[i]$ (go to 2(c)ii)
        iii. If (one of $Targets[i]$ was not a non-ignorable obstacle node)
            • Let $temp\_max\_NSI = NSI\_Dist$
        iv. Increment $NSI\_Dist$ and go to 2c
    (d) Let $CurrNode.max\_NSI = temp\_max\_NSI$
    (e) Evaluate next node (go to 2)
```

Subroutine: **Get_Distance**

Input:
- $WhiteNode$, the current *white* node
- $Target$, the current target node
- $WhiteLevel$, the level of $WhiteNode$
- $TargLevel$, the level of $Target$

Output:
- $targ\_near$, the near-distance from $WhiteNode$ to $Target$
- $targ\_far$, the far-distance from $WhiteNode$ to $Target$

```
1. If ($Target$ is *grey*)
    • Determine the *black* and *grey* child nodes of $Target$; store in array $TargChild$
    • Initialize $targ\_near$ as some maximal value and $targ\_far$ as 0
    • Let $ChildLevel = TargLevel + 1$
    • For all elements $TargChild[i]$:
        (a) Call **Get_Distance**, passing $WhiteNode$, $TargChild[i]$, $WhiteLevel$, $ChildLevel$
            and receiving $child\_near$, $child\_far$
        (b) Let $targ\_near$ be the minimum of $\{targ\_near, child\_near\}$
        (c) Let $targ\_far$ be the minimum of $\{targ\_far, child\_far\}$
        (d) Evaluate next $TargChild[i]$ (go to 1)
2. Else /*$Target$ is *black*/
    • Call **CalcNearFarDist**, passing $WhiteNode$, $Target$, $WhiteLevel$, $TargLevel$, and
      receiving $targ\_near$ and $targ\_far$
3. Return $targ\_near$, $targ\_far$
```

Subroutine: **CalcNearFarDist**

Input:

- $WhiteNode$, the current *white* node
- $Target$, the current target node
- $WhiteLevel$, the level of $WhiteNode$
- $TargLevel$, the level of $Target$

Output:

- $near\_dist$, the near-distance from $WhiteNode$ to $Target$
- far_dist, the far-distance from $WhiteNode$ to $Target$

1. (Assume we have location codes for $WhiteNode$ and $Target$) Calculate reference points $White\_ref$ and $Targ\_ref$ of $WhiteNode$ and $Target$, respectively, from location codes
2. Let $White\_size$ be side length of $WhiteNode$ node
3. Let $Targ\_size$ be side length of $Target$ node
4. Obtain the bounding box dimensions for $WhiteNode$ and $Target$:
   - Find the minimum $min.x$ and maximum $max.x$ of $\{White\_ref.x$, $White\_ref.x+White\_size$, $Targ\_ref.x$, $Targ\_ref.x + Targ\_size\}$
   - Find the difference $diff.x = max.x - min.x$;
   - If $(diff.x < 0)$, let $diff.x = 0$
   - Do the same for y and z, obtaining $diff.y$, $diff.z$
5. (Refer to Figure 7.2) Find far-distance $targ\_far$:
   - Subtract $Targ\_size$ from each of $(diff.x, diff.y, diff.z)$
   - Let far_dist be the length of the vector $diff$
6. Find $targ\_near$:
   - Subtract $White\_size$ from each of $(diff.x, diff.y, diff.z)$
   - Let $near\_dist$ be the length of $diff$
7. Return $near\_dist$, far_dist

Appendix C

Detailed ODM Collision Detection Algorithm

Main routine: **ODM_Detect**
Input: octree distance map, robot position, robot radius.
Output: collision situation: TRUE if collision, else FALSE.

```
1. Localize robot to the highest white node $W$ in tree
2. Retrieve $min\_NSI$ and $max\_NSI$ for $W$
3. For $NSI = min\_NSI$ to $max\_NSI$ do:
   (a) Generate list of nearest black and grey target nodes at same level $L$ as $W$, with nodal
       distance from $W$ equal to $NSI$
   (b) For each target node $T$ in list
       i. Call Recurse_Detect; receive collision result in $Result$
       ii. If ($Result$ == TRUE) /*collision*/
           • Return TRUE result
       iii. Else /*no collision*/
           • Go to next target node (go to 3b)
   (c) No collision has been found; increment NSI and go to 3
4. No collision has been found. Return FALSE result.
```

Subroutine: **Recurse_Detect**
Input:
- robot white node W

- robot center voxel position w
- robot radius $robot\_size$
- target node T
- level of white node, L

Output: collision situation: TRUE if collision, else FALSE.

```
1. If (T is black)
    • Calculate near-distance for robot center voxel w and T
    • If (robot_size > near-distance)
        – Collision is certain; return TRUE
    • Else:
        – No collision; return FALSE
2. Else (if T is grey)
    (a) Calculate the maximum-radius bound for w and T (see Figure 7.6)
    (b) If (robot_size > maximum bound)
        • Collision is certain; return TRUE result
    (c) Else:
        i. Recursively call Recurse_Detect with all black and grey children of T as target nodes
        ii. If any result from recursive call is TRUE
            • Collision detected; return TRUE result
        iii. Else (no TRUE results)
            • No collision with T; return FALSE
```

Appendix D

ODM Collision Detection for a Line Segment

The following algorithm can be used for performing collision detection against a line segment using an ODM. This algorithm could be useful with robots modeled as line segments instead of spheres.

```
Main routine: **Line_Detect**
*Input*: ODM, line segment endpoints $P_1$ and $P_2$, line length *len*
*Output*: collision situation: TRUE if collision, else FALSE.

1. Call ODM_Detect, specifying a sphere with center $P_1$ and radius of $0.5len$; receive collision
   result in Result
2. If (Result is FALSE)
    • Call ODM_Detect, specifying a sphere with center $P_2$ and radius of $0.5len$; receive
      collision result in Result
3. If (Result is FALSE)
    • Return FALSE collision result /*No collision with line seg*/
4. Else
    • Let $M$ be the midpoint on line $L$
    • Recursively call Line_Detect, passing $P_1$ and $M$ as endpoints of a line, and $(0.5len)$
      as the length; receive result in Result
    • If (Result is TRUE) /*Line segment in collision*/
    – Return TRUE collision result
• Recursively call **Line_Detect**, passing $M$ and $P_2$ as endpoints of a line, and $(0.5len)$ as the length; receive result in *Result*
• If (*Result* is TRUE) /*Line segment in collision*/
    – Return TRUE collision result
• Else /*No collision*/
    – Return FALSE collision result
```