Extended Octree Distance Map (EODM) : A New Representation for Collision Detection

by

María del Carmen C. Amézquita Benítez

B.A., Universidad de las Américas Puebla, México 1996

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

© María del Carmen C. Amézquita Benítez 2002
SIMON FRASER UNIVERSITY
October 2001

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: María del Carmen C. Amézquita Benítez

Degree: Master of Applied Science

Title of thesis: Extended Octree Distance Map (EODM) :
A New Representation for Collision Detection

Examining Committee: Professor Andrew H. Rawicz.
Chairperson

Professor Kamal K. Gupta, Senior Supervisor
School of Engineering Science

Professor Binay Battacharya, Supervisor
School of Computing Science

Professor John Dill, Examiner
Professor, School of Engineering Science

Date Approved: _____

Abstract

In this document we propose a new representation called Extended Octree Distance Map (EODM) for efficient collision detection in static environments. EODM constitutes a complete and systematic hierarchical representation for distance maps. Like other methodologies, it utilizes an octree as the base representation. Along the perimeter of each white node in the octree, it stores a distance function that represents the distance of each boundary point of the white node to the obstacle closest to that point. EODM is computed once and then repeatedly used for collision detection queries. We present algorithms for creating the EODM and use it for collision detection. Our experiments in 2D and 3D show that while EODM requires more memory than an octree and an ODM, it decreases collision detection time substantially.

Acknowledgments

I would like to thank all the members of the examining committee for their interest in this project.

I am extremely grateful for the continuing support and guidance of my supervisory committee: Dr. Kamal K. Gupta and Dr. Binay Battacharya. Without your observations, recommendations and dedication this work would not have been possible.

My gratitude also extends to Dr. Andrew Rawicz for his time and to Dr. John Dill for opening his door, listening and giving me an opportunity in times of confusion.

My deep appreciation and admiration for Brigitte Rabold and all the staff at the School of Engineering Science Department.

I thank Dr. Ofelia Cervántez, Dr. Warren Greiff, Dr. Juan Manuel Ahuactzin and the faculty of the Engineering Department from Universidad de las Américas Puebla (UDLA-P) for their example and encouragement.

Finally, I would also like to thank Consejo Nacional de Ciencia y Tecnología (CONACYT), Mexican National Science Foundation, for its support during this research from september 1997 to august 1999.

Dedication

To the patience, wisdom and support from my parents. For their labor of love and their warrior courage through all hardships I put them through.
To the ghosts from the past that allowed me to step on their footsteps and retrace their journey.
To the solitude and loneliness that allowed me to get to know myself.
To Reza Afrashteh, for the everlasting tea time for the soul.
To Andrea Downie, who taught me to celebrate life through her dance. For her words and encouraging spirit. For being a true teacher.
To Helen Pinto and Luis Goddyn, with whom the celebration of life became complete.
To my guardian angels Reza Naserasr and Istvan Harmati.
To the uniqueness, complicity and sisterhood of Guadalupe Delgadillo.
To the far away angel that lights my way. For being my inspiration and source of strength beyond frontiers. Thank you Gábor Vass.

To the infinite blessing of friendship that came in different languages, customs, ideas, flavors and sounds: Sigal Blay, Laura Chávez, Caroline Dayyani, Guillermo Fernández, Ian Gipson, Andrew Haskell, Julia Istkevitch, Dulce and Wuilbert Jaramillo, Kamran Kaveh, Deyra Kelly, Karen and Gaby Kwong, Ming Li, Te mei Li, Qingguo Li, Lin Lin, Shiming Liu, Tissaphern Mirfakhrai, Herbert Noriega, Helen Pinto, Babak Taati, Xiaoli Zhang and the people in SFU's International Club.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Collision detection is broadly employed in a number of disciplines. Robotics and Computer Graphics are two of the main current areas. A collision detector constitutes a key building block for path planners. The performance and efficiency of path planners is critically dependent on the number and speed of collision tests, therefore it has become a fundamental problem in Robotics [15].

Collision detection is concerned with the efficient and fast solution of the following problem: "given two objects, O_A and O_B, do they interfere?" [22].

Collision avoidance algorithms can be classified into (i) continuous motion collision detection and (ii) static collision detection [10]. The continuous problem involves moving objects. For each one of the moving objects, its position and velocity is known beforehand and the collision detection problem is to find if they collide at any instant during a given time interval. On the other hand, static collision detection "freezes" the objects at a particular time and any contact or overlap between them is detected [10]. Continuous collision detection can be discretized at a certain time resolution and treated as a sequence of static collision detections. Due to the complexity of representing the configuration space (C-space) of a robot, path planning uses instead a discretized representation of it. At each discrete point, obtained by sampling and searching the discretized C-space, a collision detection test is required [15].

Collision detection performance is strongly dependent on the representation that is used to model the environment. When complex geometric models for obstacles are used, collision detection becomes a bottleneck in simple path planning applications since they can invest hours trying to solve a problem due to the "millions" of collision tests (as demonstrated in

the Sandros system [4, 23], the work of Kavraki et al. [27] and Jung [25]). Therefore the search for more efficient approaches for collision detection is crucial in order to produce fast motion planning applications for the "real world". Most of the work on collision detection has used two broad types of representation (i) more "abstract" geometric models (mostly polyhedral in 2-D and 3-D, but also including curved surfaces) [34], and (ii) "raw" (or primitive) spatial occupancy maps, which are discretized binary bitmaps (obstacle/free) of the workspace (closer to what most common range sensors, such as stereo-vision, laser range finders, and sonars, etc. directly provide). Our motivation for this work comes primarily from sensor-based motion planning in robotics, where it is natural, easy and straightforward to represent the workspace as a spatial occupancy map [41].

It is because speed in collision detection is a key issue in efficient path planning that we have developed our Extended Octree Distance Map (EODM). EODM is a novel collision detection module for static environments that captures the distance to the obstacles hierarchically. Using an octree as a base representation, the distance is stored around the boundaries of each white node in the tree.

The following section will review five models for object representation that have been used for collision detection. A more detailed description of these approaches can be found in [12, 17, 22]. Their advantages and drawbacks for collision detection problems are also presented. In section 1.2 we discuss the research work that has influenced the development of our EODM. For recent surveys on the state of the art of 3D collision detection please see [24, 32].

1.1 Background: Object Representation for Collision Detection

The main approaches to represent objects considered are: Boundary Representations, Constructive Solid Geometry, Sweep Representations, Cell Decomposition, and Hierarchical Representations (particularly Octrees). Their characteristics are described next.

1.1.1 Boundary Representation

An object is represented as a set of bounding faces, composed by edges which are delimited by vertices. Therefore, this methodology is useful when either objects are polyhedra or they

can be approximated by a group of polyhedra. Collision detection schemes test every feature (faces, edges, vertices) for intersection and with some assumptions the test can be performed in constant time as in the well known ICOLLIDE [5, 30], RAPID [31] and VCOLLIDE [20] collision packages.

1.1.2 Constructive Solid Geometry (CSG)

CSG constructs objects from some primitives like blocks, pyramids, cylinders, cones and spheres. An object is constructed using set operators such as union, intersection, and difference. The object is represented as a tree where the operators are stored at the internal nodes and the primitives at the leaves [17]. In contrast with the Boundary Representation paradigm, CSG does not require an explicit definition of the features of the objects. The properties of the object are derived bottom-up starting first with the properties of the leaves. The main advantage of CSG is its simplicity to represent curved and more sophisticated objects with fewer parameters. However, each time a basic operation is required, the tree must be evaluated using a depth-first technique. For this reason, interference detection might lead to expensive and/or complex algorithms.

1.1.3 Sweep Representations

The result of this approach is a volume generated after moving (sweeping) an object along a trajectory in space. There are two kinds of sweeps, translational (following a linear trajectory which is also perpendicular to the 2D object) and rotational (where the object is rotated around an arbitrary axis). A general sweep is the resulting volume, (which may change in size, orientation or shape), when an arbitrary curved trajectory is used to sweep the object. The advantage of this method is that a representation can be generated for more elaborate objects, despite the complexity of their underlying analytical geometry. Checking for interference between a point robot and an object translates to checking the inclusion of the point in the swept volume.

1.1.4 Cell Decomposition

Cell Decomposition approaches represent an object as a connected net of cells. Cells can be identical (commonly squared or triangled) or they can have different sizes. In 3-D, tetrahedra are employed for the decomposition in such a way that all of them are connected

by a face, edge or a vertex. When a grid of cubes (also known as voxels or volume elements) is used, the approach is called Spatial Occupancy Enumeration, which can be represented sequentially or hierarchically (described in the next category). Cells are useful when the nature of the application takes advantage of the topological properties of the objects. Spatial Occupancy Enumeration is relatively simple to apply because each cell that is totally or partially occupied by an object has value of 1, otherwise they have a value of zero [22]. Cell Decomposition has proven to be advantageous for collision detection too, for example, in the work of Barraquand et. al [2], if a point robot falls in a cell that has a positive value, an obstacle has been found and a simple comparison betwen the radius of the robot and the cell value determines the collision/free status of the robot. Even though Cell Decomposition provides a fast performance during collision detection, it may require a very fine resolution to represent an object, and hence high memory requirements.

1.1.5 Hierarchical Representations

The main goal of hierarchical representations is to reduce the amount of storage and redundancy of the Cell Decomposition approaches. There are several hierarchical data structures such as k-d trees, sphere trees, pyramids, bintrees, striptrees, etc. Some of the most practical in collision detection are quadtrees, octrees [38, 39], and sphere trees [8, 36]. Quadtrees are a special case of the Spatial Occupancy representation. In general, each node in the hierarchy is labeled according to the portion of the object that it represents: a node is *exterior* if it covers a portion that is outside of the object's boundary, a node is *interior* if it is completely enclosed by the object. If the status of the node can not be decided, (e.g. a node overlapping the boundary of the object), it is recursively decomposed until each one of its children has a label assigned or a certain decomposition resolution level is reached. Angel del Pobil et al. [8] employ two layers of spheres to generate an internal and an external cover of the object to be represented.

Hierarchical Representations are resolution complete, thus, able to represent complex objects with accuracy according to the storage that is available. They also facilitate collision detection because the hierarchy preserves the cells in order in terms of their spatial occupancy (characterized by the branches in the tree) and their size (characterized by their depth in the tree). Moreover, they have become important and useful tools because of their efficiency of representation and their improved execution times (e.g. when a search of a spatial point is performed). Samet emphasizes that the savings during execution time

are a direct result of the data aggregation. He also affirms that hierarchical data structures provide a major advantage because they are both, easy to understand and easy to implement [38].

In sensor-based planning the robot has no *a priori* knowledge of the environment. Useful planners must rely on sensors that capture with accuracy the information from the surrounding environment as the robot moves. Two decades ago Hans Moravec and Alberto Elfes [35, 11] proposed the creation of maps to assist mobile robots to navigate in unknown environments while avoiding obstacles. The methodology was called Occupancy Grid, a matrix that represents 3D space, in which each element has a value. A positive value indicates the presence of an object, 0 indicates that the space is free. Other extensions use the values in the matrix to represent the level of confidence, (probability, certainty) that an object is occupying that portion of space.

For a robot immersed in a changing environment there are memory, computational capacity and time restrictions for planning [29], therefore, it is crucial to minimize the amount of data that is sensed and processed while the robot is in motion, storing only "salient" features of the environment. It is due to the memory limitations that an occupancy grid is expensive to compute. Hierarchical representations of the environment require less memory and may even be more efficiently obtained [41]. Because of these requirements, we believe that the data structure that best models the environment for our purposes is an octree. A brief description of octrees and their encoding is provided in the next subsection.

As we said before, because collision detection is a key component for path and motion planning, under uncertain environments like in [27, 41], it is crucial to increase the speed of collision tests. With these ideas in mind, reducing storage and increasing speed, we propose our EODM as a novel approach for efficient collision detection. In the future, we plan to use EODM as part of a practical framework that incorporates sensing for geometrical reasoning with on line planners efficiently.

1.1.6 Quadtrees and Octrees

Quadtrees and Octrees are hierarchical data structures that have been widely employed in diverse fields such as image processing, geographic information systems (GIS), pattern recognition, solid modeling, computer vision and robotics [38].

The characteristic properties of quadtrees and octrees depend on three factors: the kind of data that they represent, the criteria by which the decomposition is performed

and the decomposition resolution (number of times that the decomposition is performed recursively) [38]. The decomposition may be regular (dividing into equal parts on each level and producing regular polygons), or input dependent. The decomposition resolution may be fixed beforehand or determined by the properties of the input data. Brabec and Samet [3] provide an interesting set of Java applications of the different quadtree data structures described in [38].

The most common quadtree representation is the *region quadtree*, treated here as quadtree. Originally, they were employed to represent two dimensional binary images. A quadtree is represented as a tree in which each intermediate node has four children (a tree of degree 4). The root node corresponds to the entire binary array. If the image does not consist entirely of 1's (or 0's), it is subdivided into four quadrants of equal size. Each quadrant is recursively tested with the previous criteria. Each node in the tree represents a quadrant of the image that has been partitioned. The leaf nodes represent the portion of the image that does not require further subdivision (e.g subquadrants that contain only either 1's or 0's). All the nodes in a quadtree are labeled with a "color". The leaf nodes that represent blocks comprised totally inside a region in the image (e.g. composed exclusively by 1's) are "black". Those leaves that represent blocks located completely outside the region (composed only by 0's) are labeled as "white". All the intermediate nodes (blocks containing a mixture of 0's and 1's) are "gray". For a $2^n \times 2^n$ image, the root node is located at level n, while the leaves are located at level zero. An example of a binary image array of $2^3 \times 2^3$ and its quadtree representation is shown in Figure 1.1. The 1's correspond to elements included in the region, otherwise they lay outside. Note the advantage of the region quadtree storage v.s. a binary array.

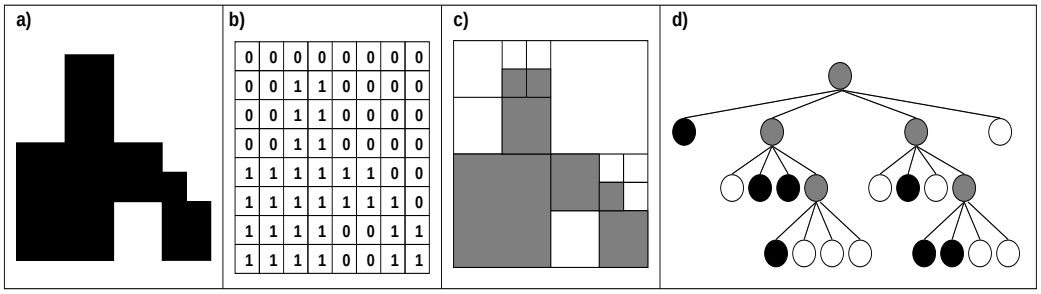


Figure 1.1: Binary array image and its quadtree: a)region b)binary array c)quadtree d)tree

The three dimensional version of a quadtree is called an octree. Given a $2^n \times 2^n \times 2^n$

object, it will be recursively subdivided into octants. The lowest resolution elements of the partition are cubes called *voxels*, which consist entirely of 0's (inside of the object) or entirely of 1's (outside of the object). The octree is represented by a tree in which each node has 8 children (degree 8). As in a quadtree, the root of the node of the octree represents the entire three dimensional object. Leaf nodes represent cubes that do not require to be further subdivided. The nodes are also color labeled with the same criteria as "white" for those cubes outside the object, "black" for those ones inside the object, and "gray" otherwise. Figure 1.2

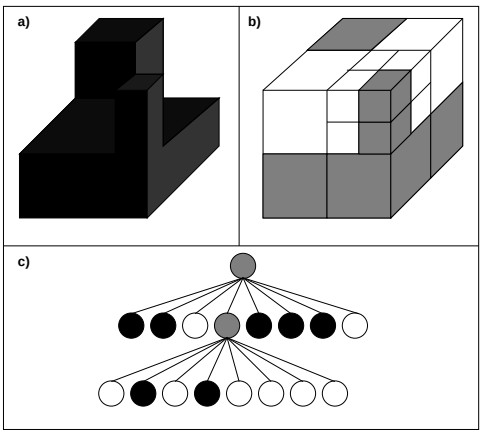


Figure 1.2: Three dimensional object and the octree representation: a)3-D object c)Octree d)tree

From this point and onwards we use the term octree to refer to quadtrees and octrees. Octrees save storage space by decomposing an image/object into homogeneous and disjoint d-cubes, (where d is the dimension), centered at predetermined positions. Octrees are employed primarily because the decomposition has the following properties:

- the partition pattern is repetitive, thus extensible to objects of every size
- the resolution of the partition can be refined in order to obtain finer patterns
- the resulting d-cubes possess the same orientation and can be mapped into each other using translations

Quadtrees and octrees can be represented as a tree or as a code list. A tree representation generates overhead because the gray nodes and a pointer to each of their children must be stored. Tree representations attempt to balance the computational cost of sequential and random access when a search is performed. Samet [38] demonstrates that the tree

representation of an octree with B black and W white blocks requires $4(B+W)/3$ nodes at most. The bound represents a considerable advantage compared to the binary array representation that will increase the storage requirements exponentially as the dimension of the environment increases. However, the worst case for an octree occurs when the aggregation of the space to be represented is minimal, (e.g. a chessboard pattern). There are other data structures that reduce the overhead inherent to octrees, like bintrees, k-d trees, etc. For more information on these methodologies refer to [37, 38]. The amount of an octree storage space depends on the resolution, (reflected in the number of levels of the tree representation), and the perimeter in 2D or surface in 3D of the object that is being represented [21, 22, 33, 38]. The experimental results of our research in Chapter 4 hold this characteristic.

On the other hand, octree list representations are not so good when random access is necessary in order to perform a search. However, they are useful because they reduce the overhead of storing pointers between parent nodes and their children. Within this group, some approaches consider the octree as a list of leaf nodes (e.g. *Locational Codes (LC)*), and others describe it as a preorder traversal of the tree (e.g. *DF-expressions*) [37, 39].

The Locational Code (LC) is sequence of digits, whose values represent a path that allows locating a leaf node starting from the octree's root. Each node is labeled by a base 4 number in the case of quadtrees. Correspondingly, base 8 is used for octrees. Each digit of the list indicates the branch of the tree to traverse. Figure 1.3 shows the quadtree block representation and the path in the tree, (shown in bold), that is necessary to traverse in order to find node N with LC: 2301_4. First, branch 2 at level one is followed starting at the root of the tree (level zero). Then successively branch 3 at level two, branch 0 at level three, and branch 1 at level four are chosen. A similar procedure is performed for octal chains. To keep compatibility with the work of Jung [26] our implementation stores the Locational Code of a node along with its level in the tree. However, the level of a node can also be derived from the number of digits that compose its LC.

Locational Codes also encode the Cartesian coordinates of each node in an octree [26, 39]. For example, in LC: 2301_4 of node N, the digits in the chain will be examined from left to right as shown in Figure 1.4. Each digit is transformed to its binary equivalent, producing a subchain. The concatenation of subchains originates the following sequence:

$$010_2\ 011_2\ 000_2\ 001_2$$

In order to obtain the z-coordinate of node N, the first digit of each binary subchain

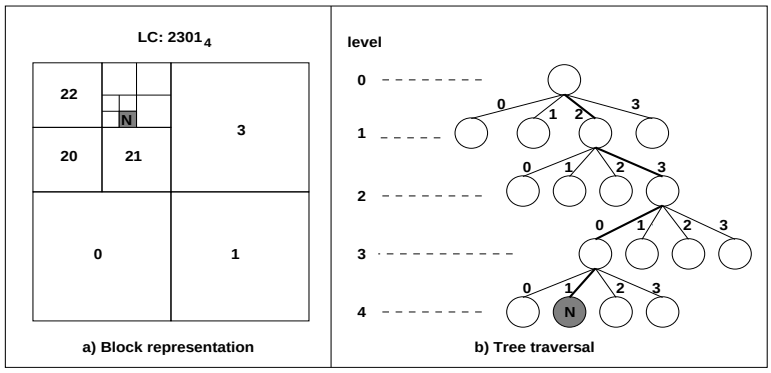


Figure 1.3: Locational Code and the traversal of the tree

is chosen and the new subchain b_z is formed with a value of 0000. b_z is then transformed to its decimal equivalent resulting in $0000_2 = 0_{10}$, thus, the zth coordinate of N is zero, (a quadtree). The subchain for the y-coordinate, b_y is obtained in a similar fashion, $1100_2 = 12_{10}$. And finally, the x-coordinate from subchain $b_x = 0101_2 = 5_{10}$. The complete set of coordinates (bx, by, bz) define a *base point* for each node in an octree, which corresponds to the leftmost and bottommost vertex. The base point of N is (5, 12, 0). In order to get the LC of a node starting from its coordinates the same process is performed but in reverse order. Notice also in Figure 1.4 that the space considered is non negative.

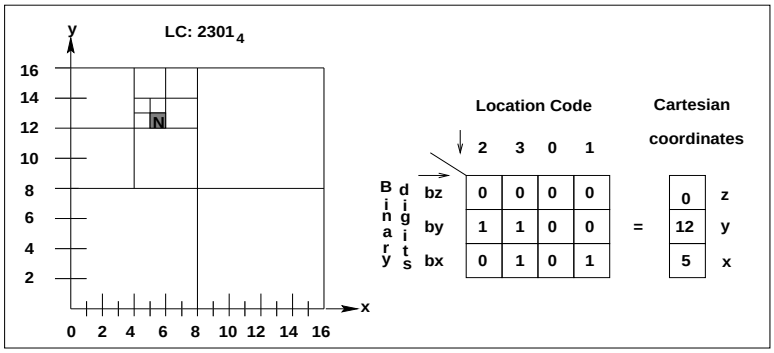


Figure 1.4: Obtaining Cartesian Coordinates from a Location Code for a node N

Location Codes are employed in our work in order to (i) retrieve the closest black node (or nodes in the case they all possess the same distance) to the center of our robot after the collision detection test (ii) preserve the adjacency between the white nodes and a structure that will be introduced in Chapter 2 called "separators".

1.2 Related Work: Collision Detection with Spatial Occupancy Representations

In order to speed-up collision detection some researchers have used discretized distance maps (DM) [2, 14]. A DM is essentially a spatial occupancy map, however, each free cell, rather than storing binary obstacle/free values, stores the distance to its closest obstacle cell. Cells that overlap with an obstacle have zero value associated with them. These distances are easily computed with a "wavefront" that expands outwards from the obstacles' boundaries [2]. Figure 1.5 shows the DM corresponding to an environment and the L_1 distances associated with each cell.

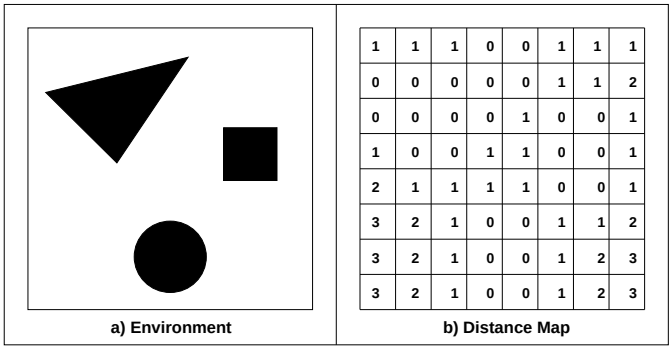


Figure 1.5: An environment and it's associated L_1 distance map

Barraquand et al. [2] used the DM to detect collisions with a given line segment (a link of the planar robot) of length l by checking if the cell corresponding to an endpoint of the line segment has a distance value $> l$. If it does, then there is no collision; otherwise the line segment is recursively subdivided into two halves of length $l/2$, and the process is repeated.

Greenspan et al. [14] used a spherical representation to cover the robot as well as A. del Pobil [7] and Quinlan [36], where each sphere is tested for collision in a systematic way. Checking a sphere for collision is a constant time operation, a simple comparison of the radius of the sphere with the distance value stored in the cell in which the sphere center lies. Figure 1.6 shows a 2D DM of an environment in which a diamond-shaped, (Manhattan), robot lies. The cells have been filled with a wavefront expansion that also uses Manhattan distance. Assume the radius of our robot is of 3 units, which is greater than the value of the cell in which the center of the robot falls, therefore the robot is in collision.

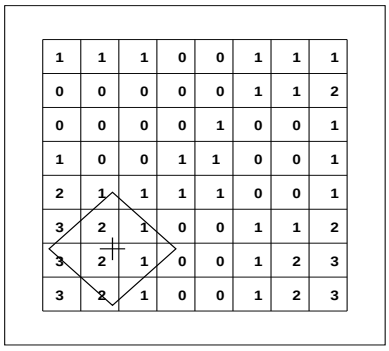


Figure 1.6: Collision detection for a 2-D distance map

DMs, however, have a significant disadvantage. The memory requirements are $O(n^d)$, where d is the dimension of the workspace and n is the number of cells along one dimension. In 3D, for instance, a workspace of 2m, with a 1mm resolution would require 8 gigabytes of memory. Larger workspaces and/or finer resolution, would increase the memory requirements significantly. Octrees, as discussed before, are a better and more memory efficient spatial occupancy representation [37]. The memory requirements for an octree are proportional to the "surface area" rather than the "volume", hence it would take $O(n^2)$ to represent a 3D environment. It is, however, a "binary" representation, i.e., a node is either free or represents an obstacle, or mixed if neither.

Related work that employs octrees for collision detection can be found in [1, 16], where the environment is modeled as an octree and collision testing consists in checking if a point is contained inside an obstacle node of the octree. Hayward [16] originally represents the links of a manipulator as a set of hemispheres and cylinders. Then an octree is created for each link. The obstacles of the environment are grown employing the radii of each robot component, and thus, "reducing" the robot to line segments. Collision detection is a recursive process that partitions each line segment into two and tests each subsegment for interference with the corresponding octree. The process is memory intensive. The main disadvantages of this approach are that it is necessary to store and keep track of several octrees at a time and that the octrees must be recomputed for each new configuration of the robot. Additionally, the search for collisions is an intensive process, specially if we consider a dense domain because at each configuration the algorithm tests an octree against another.

Another methodology proposed by the same researcher utilizes a single octree and control points distributed over the surface of the robot. Each control point is tested against the

octree representation. If any control point lies inside an obstacle node, a collision has been found. Despite the "simplicity" of this approach, it is important to consider both the location and the number of control points that have to be placed in order to ensure an efficient collision detection scheme, facts that compromise the robustness of the model. Moreover, contact situations between the robot and an object may be difficult to detect, for example when the object is completely included inside the robot's boundary representation.

The main problem with these octree approaches is their lack of distance information, how "far" are the obstacle nodes with respect to the free nodes. To remedy this absence of knowledge, Hierachical Distance Transforms have been proposed [38, 40]. They have been used in image processing to provide a certain sense of proximity to obstacles, however, the aim is achieving efficient image representation. The *distance transform* is the shortest distance from the center of an obstacle node to the boundary of a free node. For each obstacle node in a quadtree, its eight-neighbors are tested in order to determine the L_{inf} distance to the closest free (white) pixel to that particular obstacle node. This approach invariably stores a *single* distance value for the entire quadtree node, which has been proven effective for image representation but insufficient for collision detection [25]. The work of Schneier [40] proposes a manhattan (L_1) distance transform that stores the minimum distance to a white pixel in each one of the directions (north, east, south, west) around the obstacle node.

Jung [25] proposes a new representation of the environment called Octree Distance Map (ODM). The main difference between the work of Schneier and Jung is that the ODM computes distances for white (free) nodes, not obstacle nodes, and that it constitutes the first step towards capturing this distance information in a hierarchical manner. ODM adapts and augments the conventional octree by storing a minimum and a maximum distance values (2 bytes) for each white node of the octree. These values are called Nodal Separation Index (NSI). The minimum_NSI represents the minimum distance between any point in the white node to the obstacles. This implies, that a spherical robot of radius less than the minimum_NSI is guaranteed to be collision free if its center is located in that white node. The maximum_NSI represents the lower bound of the robot's radius such that no matter where the robot is located in the white node, it is certain to be in collision with an obstacle. The ODM collision detection algorithm uses the minimum and maximum NSI's of the white node to limit the search (as compared to an octree) for obstacle nodes that can result in collision. ODM therefore, trades memory for speed of collision detection when compared to an octree. Moreover, ODM stored minimal distance information, just two values.

1.3 EODM: Basic Idea

This thesis started with a simple basic question: "what distance information can be stored in an octree to further improve collision detection efficiency without seriously affecting storage?". Our work shows that the information we need is a distance function defined over the boundary of the white nodes that represents the closest obstacle distance to each point on the boundary. Formally speaking, this distance function is obtained from the intersection of the Voronoi diagram of the free space with the boundaries of the white nodes. We show that with this information, collision detection (for a spherical robot) is a constant time operation for L_1 metric, essentially a look up in the stored distance.[1] As a result, EODM significantly decreases the collision detection time when compared to other octree based approaches, however, it is a bit more memory intensive than an octree, but still considerably less than a bitmap representation. On average, the storage depends only on the area of the white nodes and not on the volume, although the worst case scenario will still be proportional to the volume.

The main philosophy behind our research is to find intermediate approaches between a Distance Map (with small run-time for collision detection but high requirements of memory) and a conventional octree (with relatively long run-time but low memory requirements). This memory versus collision detection time trade-off for DM, EODM, ODM and octree, is qualitatively shown in Figure 1.7. From our point of view, EODM constitutes an option that offers a promising memory-runtime compromise.

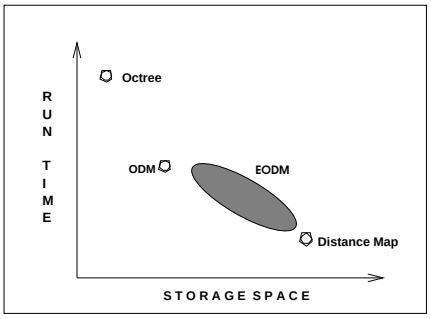


Figure 1.7: Memory-Runtime trade-offs. We expect EODM to lie somewhere in the shaded region

[1] Given the location of the center point of a spherical robot, it takes $O(logN)$, N = number of leaf nodes, to locate the node in which the robot center lies, for any hierarchical representation be it octree, ODM or EODM. Once the node is located, the time taken to detect collision is constant for EODM. This is where the time saving comes compared to octree or ODM.

We must cite here the work of Hoff et al. [19], developed simultaneously as our research progressed. They presented a hardware implementation to compute discrete generalized Voronoi diagrams and used it successfully for fast dynamic path planning. They generate a coarse polygonal mesh that approximates the obstacles as a preprocessing step. The 3D Voronoi diagram of the obstacles is discretized into a set of planar Voronoi diagram computations. Each discrete "slice" of the Voronoi diagram is generated for a set of arbitrary obstacle points (sites). The distance to the closest site is stored in a hardware version of a Z-buffer. The performance of the system is bounded by the speed with which the buffer rasterizes the pixels of the scene. [18] Their work, as well as ours, reinforces strongly the advantage and the need of modeling efficiently the notion of distance using Voronoi diagrams.

The thesis is organized as follows. Chapter one provides a brief introduction to obstacle representation and the background work that supports our research. Chapter two describes our collision detection approach for a 2D case, meanwhile Chapter 3 provides a deep description of the extension to the 3D problem. Chapter four provides the experimental results obtained, in terms of performance and memory employed. It also explores the relevance of our approach compared with other classical ones, like a Distance Map, and an octree. We conclude in Chapter five with a discussion of future possibilities for our approach. Some relevant algorithms that we have designed and implemented and the user's manual of our implementation can be found in the Appendices sections.

1.4 Contributions of the Thesis

The contributions of our research are:

- A new octree-based representation for collision detection called Extended Octree Distance Map (EODM) that trades off memory for faster collision avoidance than that with an octree.
- Constant time collision detection is acurate up to the resolution of the underlying octree. This assumes that the white node in which the center of the robot falls has been already found.
- Mathematical models and algorithms for constructing an EODM.
- Mathematical proofs and collision detection algorithms with EODM.
- Empirical results showing that EODM speeds up collision detection dramatically when compared to other octree-based methodologies.

Chapter 2

Extended Octree Distance Map (EODM): 2D Case

In this Chapter, we present the 2-dimensional version of our Extended Octree Distance Map (EODM). The generalization to the 3D problem is relatively straightforward, though somewhat tedious in detail (as discussed later in Chapter 3). For brevity, we will use the term octrees to refer to both quadtrees and octrees.

2.1 Assumptions and Notation

We assume a static known environment and that an octree model of the environment is given. A white node of an octree, denotes that the entire node is in free space, and a black node denotes that the entire node is part of an obstacle(s), and of course, a grey node denotes that part of it is free and part lies in an obstacle. A white node of the octree is denoted by W. The metric considered is L_1 (Manhattan metric) and we use $d(A, B)$ to denote the L_1 distance between two points A and B.

Let $\mathcal{R}$ denote a manhattan spherical robot (diamond shape) with radius r. The robot is located at a point L, if its center is positioned on L. A collision query consists of the following question: given a spherical robot $\mathcal{R}$ with radius r, and location L, determine if it is in collision with the obstacles in the environment. Note that the robot location as well as the robot's radius may vary from one query to another.

2.1.1 Distance

For the purposes of our research, the present work considers Manhattan distance (L_1) to measure the proximity between a white node in an octree to its closest obstacles. Manhattan distance is defined as:

$$L_1(\text{a,b}) = \sum_{i=1}^{d} \|(a_i - b_i)\|$$

where d is the dimension of the space.

2.1.2 Motivation

Since the robot location L can vary within a white node, quite possibly a reasonably large region, the vital purpose of our research is to determine "what distance information can be stored to aid in collision detection?". The key idea behind EODM is to store the distance around the "boundary" of a white node W to its closest obstacles. A schematic is shown in Figure 2.1. The left side shows a white node surrounded by obstacles (black regions). The horizontal axis in the graph on the right is the length traversed (denoted by parameter s) along the boundary of the white cell starting from vertex 0. The vertical axis represents the distance of the corresponding boundary point to the closest obstacle. This distance function along the "surface" (perimeter in 2D) $d_p(s)$ encodes all we need to know about the proximity of obstacles. This will require additional storage, however, it is still proportional to the "surface" and not the "volume" of the free region.

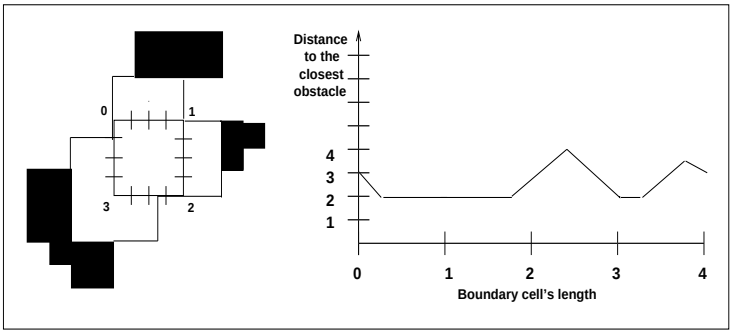


Figure 2.1: The perimeter distance function, $d_p(s)$

We show below that for the L_1 metric, $d_p(s)$ is sufficient to carry out the collision query in *constant* time, once the white node in which the robot center lies has been found; it takes $O(logN)$ to locate the node for any hierarchical representation be it a simple octree, ODM

or EODM.

2.1.3 Constant time collision detection with $d_p(s)$

Consider Figure 2.2.a. Let L be the center of $\mathcal{R}$ (diamond shape) lying in a white node W. Let S be one of the boundary edges of W. Let P be the projection of L onto S. Let A be a closest obstacle point (say, from obstacle $\mathcal{O}$) to S on the left side of S. This implies that no other obstacle point exists strictly within the L_1 sphere (striped diamond shaped region shown in Figure 2.2b) centered at L and radius $d(L, A) = x + y + l$, with x and y as shown in Figure 2.2.

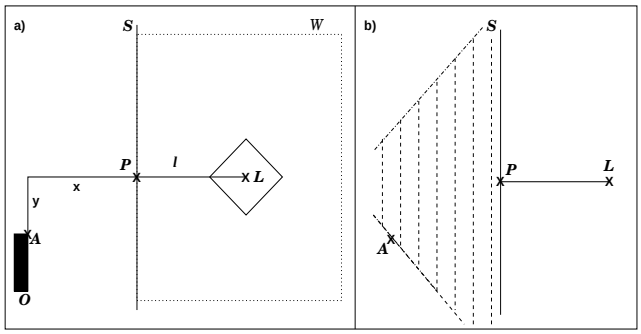


Figure 2.2: $d_p(s)$: the shortest distance between any point in W and an obstacle point

Let the distance between P and L, $d(P, L)$, be l. The shortest L_1 distance between L and A is given by the expression $l + x + y$. Note that there may be many different paths with the same shortest distance; in fact, all monotonic (in both horizontal and vertical coordinates) paths from A to L are shortest paths by the very property of L_1 metric. However, we are not interested in all the possibilities, only in the path between L and A that passes through point P as depicted in Figure 2.2.a.

Theorem 1, below, formally shows that a closest obstacle point (on the left side of S) to the robot's location L is also a closest obstacle point to the projection of P onto S. Since P belongs to S, we can easily compute the length parameter along the perimeter of S (say, it is s_P) and get the closest obstacle distance to P simply by evaluating (or indexing into) $d_p(s)$ at $s = s_p$. The closest obstacle distance to L, denoted by $d_O(L)$ is then simply the sum of the two, the distance from L to the projection point P plus the distance value stored at P, i.e.,

$$d_O(L) = d(L, P) + d_p(s_P) \tag{2.1}$$

If the robot's radius $r \geq d_O(L)$, the robot is in collision, otherwise it is not. Clearly this simple test is performed in constant time.

Theorem 1 Let point P be the projection of point L onto S. If A is a closest obstacle point to P, then it is also a closest obstacle point to L.

Proof

Our proof is by contradiction. See Figure 2.3 for an illustration. Assume that exists another point $A^{'}$ (from the same or another obstacle $\mathcal{O}^{'}$) strictly closer (than A) to L, such that $d(L, A^{'}) = x^{'} + y^{'} + l$ (a monotonic path, hence the shortest). This means that:

$$\Rightarrow dist(A^{'}, L) < dist(A, L) = x + y + l$$

But, we know that $dist(A', L) = x' + y' + l$

$\Rightarrow x' + y' + l < x + y + l$

$\Leftrightarrow x' + y' < x + y$

$\Rightarrow A^{'}$ is closer to P than A which is a contradiction. $\diamond$

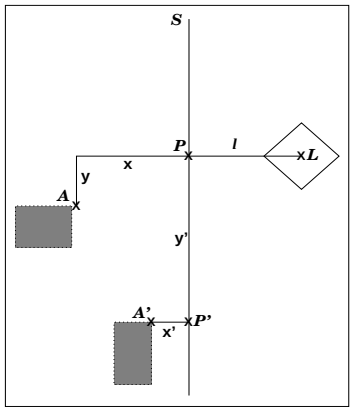


Figure 2.3: Proof of Theorem 1

Note that, had we chosen to work with L_2 distance, Theorem 1 would not hold. See for example Figure 2.4. The shortest L_2 paths from the obstacles to the robot are shown in dashed lines. The shortest path between point L and the closest obstacle point does not necessarily pass through the projection P of L onto S. In this case, in order to obtain the closest obstacle it would be necessary to compute the distance from L to *all* the obstacle

points that are closest points along the boundary of the node that contains the robot (parameterized by s), and select the one with the minimum value, (as shown in Equation 2.2). Thus, collision detection would not be a constant time operation.

$$d_O(L) = \min_s\{d(L,s) + d_p(s)\} \tag{2.2}$$

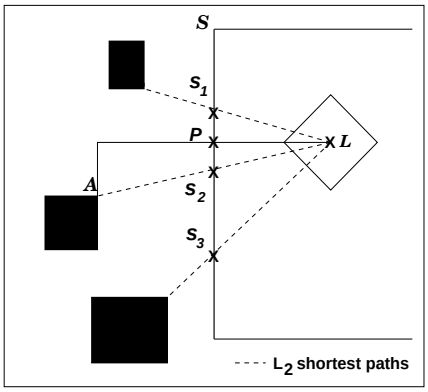


Figure 2.4: Disadvantages applying L_2 metric

2.2 2D Collision Detection

In this section we present and describe our collision algorithm for a 2D EODM called `Detect_2D_Collision()`. The corresponding pseudo-code is presented below. The procedure accepts as input the 2D EODM and the robot information (center and radius). The result is the COLLISION/FREE `Status` of the robot. In case a collision is found, the `Closest_obstacle` and its manhattan distance (`Min_dist`) to the robot are also returned as part of the output. `Min_dist` is initialized with an arbitrary large value.

Based on our previous discussion, collision detection with an EODM is a rather simple process once the node (black or white), in which the center of the robot, L, is located has been found. The search process is performed by procedure `find_node()` and takes $O(logN)$, where N is the number of leaf nodes in the octree. If this node is black, the algorithm immediately returns a COLLISION (lines 2-5 pseudo-code, shown below). Otherwise, let the node be identified as W, shown in Figure 2.5.

For each boundary edge E_i of W, we obtain the projection point P of L onto E_i using procedure `project_robot()` and then, we test P for collision using the subroutine `Detect_Collision_in_edge()`, (lines 6-8). If a COLLISION is found while the projection

onto a particular edge E_i is being examined, the `Status` of the operation is returned (lines 9-10). Otherwise, the robot is FREE of collision.

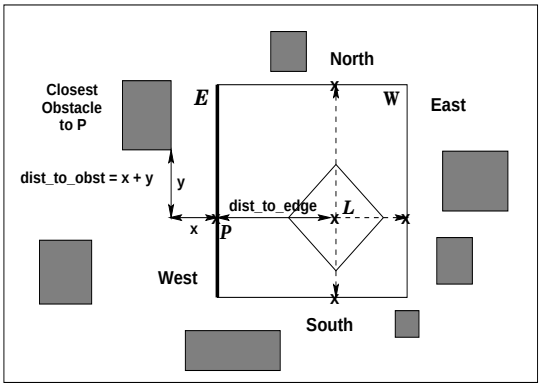


Figure 2.5: Collision Detection for a 2D EODM

```
Detect_2D_Collision( EODM, robot, Min_dist, Closest_obstacle )
Input :  EODM, Robot
Output:  COLLISION/FREE status, Closest_obst, Min_dist

1.  W = find_node( EODM, robot->center );
2.  if ( COLOR(W ) == BLACK )
3.      Closest_obstacle = W
4.      Min_dist         = 0
5.      return COLLISION

6.  For each E_i of W
7.      P        = project_robot(robot, E_i)
8.      Status   = Detect_Collision_in_edge(E_i, robot, P, Min_dist, Closest_obst)
9.      if (Status)
10.             return COLLISION
11. return FREE
```

In order to describe how a collision is tested for each edge of a white node, let `edge_dir` be the direction in which L is currently projected, (north, east, south, west). In Figure 2.5, we show P_{west}, which is the projection of L onto the *west* edge of a white node W. Let $d_O(L_{edge\_dir})$ represent the distance between the robot and the closest obstacle to an edge E_i of W in direction `edge_dir`. $d_O(L_{edge\_dir})$ is given by (see also Equation 2.1):

$$d_O(L_{edge\_dir}) = d(L, P_{edge\_dir}) + d_p(s_{P_{edge\_dir}}) \tag{2.3}$$

Then, we have that the shortest distance between the robot center L and the closest 2D obstacle, $d_O(L)$, is:

$$d_O(L) = \min_{edge\_dir} \{d_O(L_{edge\_dir})\} \tag{2.4}$$

where $$edge\_dir = \{\ north \mid east \mid south \mid west\ \}$$

The algorithm in charge of testing a collision for a particular edge of a white node is called `Detect_Collision_in_edge()`, and its pseudo-code is presented below. The first term of Equation 2.3 is represented by `dist_to_edge`, the manhattan distance from the robot center to the projection P_{edge_dir}. The second term is represented by `dist_to_obst` (see Figure 2.5). Our current implementation has a small variation (lines 2-3 pseudo-code shown below). Rather than storing actual distances along the edges of a white node, we store the interval `I` over which an obstacle point is the closest to the edge and a pointer to that obstacle point. We call this function $o_p(s)$. The actual distance function $d_p(s)$ is easily computed from $o_p(s)$ at collision detection time using procedure `compute_distance_to_obstacle()`. Adding both values, `dist_to_edge` and `dist_to_obst`, we obtain $d_O(L_{edge\_dir})$, which is represented by `dist_to_clst_obstacle` in line 4 of the pseudo-code. Line 5 performs the collision test in itself by comparing the radius of the robot and $d_O(L_{edge\_dir})$ in order to refurn the COLLISION/FREE status.

```
Detect_Collision_in_edge( Edge, robot, P, Min_dist, Closest_obst )
Input :  Edge, robot, P
Output:  COLLISION/FREE status, Closest_obst, Min_dist

1.  dist_to_edge               = manhattan(P, robot->center)
2.  I                          = find_2D_interval(Edge, P)
3.  dist_to_obst               = compute_distance_to_obstacle(I, P)
4.  dist_to_clst_obstacle      = dist_to_edge + dist_to_obst

5.  if ( robot->radius >= dist_to_clst_obstacle )
6.          if (dist_to_clst_obstacle < Min_dist)
7.                  Min_dist = dist_to_clst_obstacle
8.                  Closest_obst = I->obs
9.          return COLLISION
10.  else return FREE
```

To recapitulate, EODM stores $d_p(s)$, which is the perimeter function defined around the boundary edges of a white node. It represents the distance of each boundary point of the white node to the obstacle closest to that point. For this reason our EODM is more memory intensive than a conventional octree, however, it significantly decreases collision detection time given its simplicity (a simple comparison performed) for each collision test. So far, we have assumed that the $d_p(s)$ of each white node in the octree has been computed beforehand. The following sections present the construction of a 2D EODM in detail.

2.3 2D EODM Creation

In this section, we now describe how the EODM is created from an octree. Each time a grey node is split, we store the dividing lines (called separators from now on). First, the separators are constructed and stored in a separate data structure, `Separators_tree`. A sweep-line type algorithm is then applied to divide each separator into a set of intervals such that for each interval, the closest obstacle point remains the same. This process is repeated for every separator in the octree. For each white node, we determine the portions of the four separators that form its boundary and link the intervals to the white node boundaries. At the end of this step, each white node in the octree has an $o_p(s)$ associated with it.

2.3.1 Computing the intervals along the separators

We illustrate the intervals computation with an example, graphically shown in Figure 2.6. In this case, separator S overlaps with the east edge of W. The free space inside W and the obstacles are located on opposite sides of S. We are interested in partitioning S into intervals such that the closest obstacle point remains the same along this interval. Formally speaking, this corresponds to computing the intersection of the Voronoi diagram of the free space with the separator S. Clearly this would be a set of intervals. Our approach directly computes this intersection without having to explicitly compute the Voronoi diagram, using instead a sweep-line algorithm. In order to compute the distance intervals we consider only those obstacles lying in the outward normal direction from the boundaries of the white node. Because of the simple shape of the octree nodes (squares with axes parallel to the coordinate axes), there are great simplifications[1]. It suffices to consider only those endpoints

[1] Our approach can easily be generalized to the case where the obstacles are convex polygons.

of the edges (or portions thereof) that are x-visible from S[2]. This process is carried out by the function `obtain_black_nodes()`. In our example in Figure 2.6, one of the endpoints of obstacle $\mathcal{O}_A$ is therefore eliminated. The routine for extracting the endpoints from the visible obstacle edges is called `obtain_black_endpoints()`.

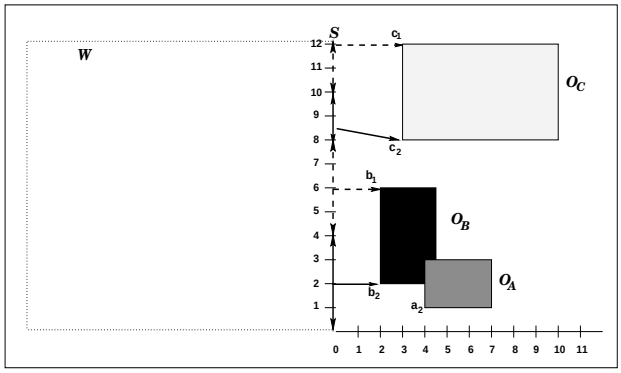


Figure 2.6: The closest obstacle function $o_p(s)$: vertex b_2 is closest to S in the interval (0,4), vertex b_1 is closest in the interval (4,8), vertex c_2 is the closest in the interval (8,10) and vertex c_1 is closest in the interval (10,12)

The obstacles' endpoints are sorted by their L_1 distance to the origin of the coordinate frame associated with the separator. Let `Black_Endpoints` be the sorted list of endpoints. Figure 2.7.a shows the `Black_Endpoints` corresponding to the obstacles in Figure 2.6. Next, we use a sweep-line with 135° orientation[3]. Each time the sweep-line hits a vertex, we compute the intersection of the sweep-line with the separator S. We keep track of two consecutive hits and the corresponding intersection points. Let `current_point` denote the top element in `Black_Endpoints`. In our example it is b_2, belonging to obstacle $\mathcal{O}_B$. Since this is the first point, an interval is opened, starting at the origin of the separator with a pointer to b_2. Let `next_point` be the next element in the list. In our example it is a_2, belonging to obstacle $\mathcal{O}_A$. Let the intersection point related to `current_point` be `current_ip`, and that corresponding to `next_point` be `next_ip`.

Somewhere between `current_ip` and `next_ip`, along the separator, the closest obstacle to S may (or may not) switch from $\mathcal{O}_B$ to $\mathcal{O}_A$. The exact point on the separator where the switch occurs, if it occurs, can be easily determined by computing the intersection of S with the L_1 Voronoi edge between `current_point` and `next_point`. The Voronoi edge is

[2] A horizontal line segment joining the obstacle edge point and S does not hit another black node.
[3] The orientation of the line depends on the quadrant in the plane to which the white node belongs.

obtained by comparing the coordinates between the `current_point` and `next_point`. Two cases can arise. If the difference of the endpoints coordinates in x (dx) is greater than their difference in y (dy), the shape of the Voronoi edge is that of Figure 2.7.b. Otherwise the shape of the edge is shown in Figure 2.7.c. When dx = dy we use by default the edge in Figure 2.7.b for a vertical sweep (S is vertical) and the edge in Figure 2.7.c for a horizontal sweep (S is horizontal), although both cases can be applied.

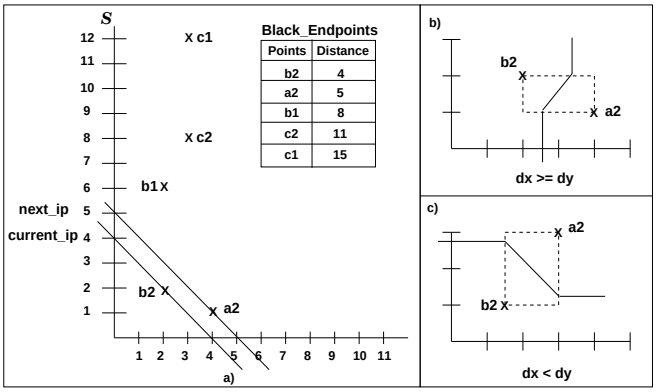


Figure 2.7: Computing the intervals

If the Voronoi edge intersects a vertical separator S (as in Figure 2.7.c), it implies that at the intersection point, the closest obstacle point switches from b_2 to a_2. In this case, we close the interval (at the intersection point) for obstacle $\mathcal{O}_B$ and open a new one (at the same point) that is associated with a_2 (obstacle $\mathcal{O}_A$). If the Voronoi edge does not intersect the separator S, it implies that b_2 is closer than a_2 to all points in S, hence we discard point a_2, keep the interval for b_2 ($\mathcal{O}_B$) open, keep b_2 as the `current_point`, and get the next point from the list as the `next_point`. In this example, the Voronoi edge is that of Figure 2.7.b, hence a_2 is discarded and the current top point from the list, b_1, becomes `next_point`. The process continues until all the points have been visited. We call this function `sweep_line()`.

Algorithm `sweep_line()` is shown next. Lines 2 and 3 initialize the state of S by opening the first interval. `sweep_dir` indicates the direction in which the sweep is performed (north, south, east, west) and is used to find the `origin` point of S. The rest of the algorithm performs the sweep-line competition by comparing the `current_point` and `next_point` until the list of `Black_Endpoints` is empty. The Voronoi edge between this two points is computed by procedure `compute_V_edge_for_two_points()`, (line 6), which is described in

more detail in section 2.3.2. Point `VP` represents the intersection point between the line separator S and the Voronoi edge of `current_point` and `next_point`. If VP exists, the intervals of S are modified accordingly by procedure `update_interval()`.

```
sweep_line( S, Black_Endpoints, sweep_dir )
Input:  S, Black_Endpoints, sweep_dir
Output: S containing the distance intervals
point origin /* point that opens the first interval */
point current_point, next_point
point VP /* intersection point between line separator and Voronoi edge */

1.  current_point = Black_Endpoints
    /* opening first interval */
2.  choose_origin(sweep_dir, S, origin)
3.  open_interval(S->Interv, origin)
4.  while (not_empty(Black_Endpoints))
5.      next_point = Black_Endpoints->next
        /* possible change in the interval:  compute Voronoi edge */
6.      compute_V_edge_for_two_points( S, current_point, next_point,sweep_dir, VP )
7.      if (VP has been found)
8.              update_interval(S->Interv, VP)
9.      current_point = next_point
```

2.3.2 Computing the Voronoi Edge between two endpoints

Determining when to open or close a new interval for a line separator is an important aspect of our `sweep_line()` algorithm. The exact point in which a new interval is created is determined by the intersection of the Voronoi edge between two endpoints and the line separator S. We mentioned before that we do not compute the Voronoi edge explicitly, however, there are some factors that we take into account. Assume that the two endpoints currently examined by the sweep-line are `a` and `b`. The points may belong to the same obstacle. The possible configurations of the endpoints while the line sweeps are shown in Figure 2.8. The dashed lines are used only to show the relationship between dx and dy. The procedure that classifies each kind of case according to the endpoints position is called `determine_point_interaction()`.

The subroutine that computes the intersection point (`pint`) between the Voronoi edge of two endpoints and S is called `compute_V_edge_for_two_points()`. It is invoked internally by

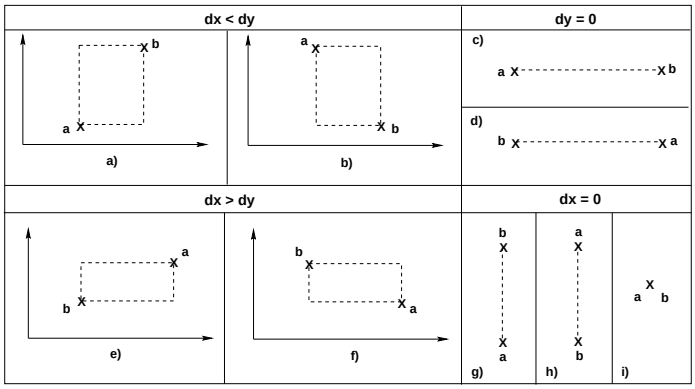


Figure 2.8: Configuration of the endpoints during the Voronoi edge computation

the `sweep_line()`. The pseudo-code is presented below. The input consists of the separator line, the two endpoints and the direction of the sweep (`sweep_dir`).

```
compute_V_edge_for_two_points( S, a, b, sweep_dir, pint )
Input :  S,points a and b, sweep_dir
Output:  pint

1.  dx = abs( a.x - b.x )
2.  dy = abs( a.y - b.y )
3.  case = determine_point_interaction( a, b )
4.  pint = compute_intersection( S, a, b, case, sweep_dir, dx, dy )
5.  return pint
```

Procedure `compute_intersection()` computes `pint`. It distinguishes four cases, either dx or dy is equal to 0, dx<dy and dx>dy. That sole relationship is enough to define the shape of the Voronoi edge. We examine each case next:

- If $dx = 0$, the Voronoi edge for cases in Figure 2.8.g and Figure 2.8.h is horizontal and passes exactly through the middle point of the dashed line that joins the endpoints,(see Figure 2.9). When the sweep is performed for a separator S in the horizontal direction there is no intersection between the Voronoi edge and S. Therefore, `pint` in the above pseudo-code is NULL. For the case in Figure 2.8.g, point `a` is the closest to S (see Figure 2.9.a). In Figure 2.8.h, point `b` is the closest to the horizontal separator (see Figure 2.9.b).

When the sweep is vertical, `pint` exists and the intervals are modified accordingly. For Figure 2.8.g we close the current interval and associate it with endpoint `a` as the closest one, and open a new interval associated with point `b` (see Figure 2.9.a). The opposite happens for Figure 2.8.h.

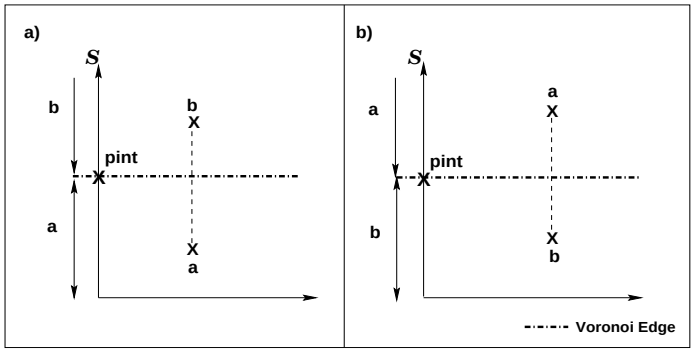


Figure 2.9: Voronoi edge when $dx = 0$

- When $dy = 0$, (Figures 2.8.c and 2.8.d), the Voronoi edge, illustrated in Figure 2.10, is a vertical line that cuts through the midpoint of the dashed line between the endpoints. If the sweep is performed in the vertical direction, there is no intersection with S, (`pint` is NULL). However, point `a` in Figure 2.8.c is kept as the closest to S, (see Figure 2.10.a). Point `b` in Figure 2.8.d is kept as the closest one in Figure 2.10.b. For a horizontal sweep, `pint` is computed and the intervals are modified, e.g. we close the current interval which is associated with point `b` in Figure 2.8.d and open a new one for endpoint `a` (see Figure 2.10.b).

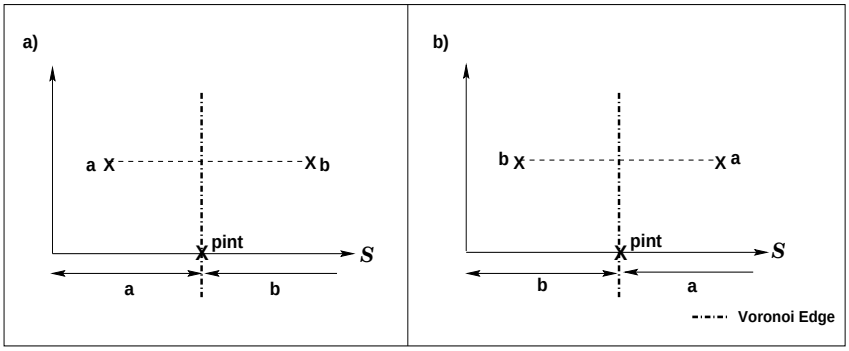


Figure 2.10: Voronoi edge when $dy = 0$

If a horizontal sweep is performed for Figure 2.8.i, we proceed as when $dy = 0$, otherwise, we find the intersection as when $dx = 0$.

- For $dx < dy$, examples are shown in Figure 2.8.a and Figure 2.8.b. Their Voronoi diagrams are shown in Figure 2.11.a and Figure 2.11.b respectively. We use Figure 2.11.a to describe the computation of the Voronoi edge. We apply a "squaring" process that evens the difference of length between dx and dy (see dashed squares in Figure 2.11).

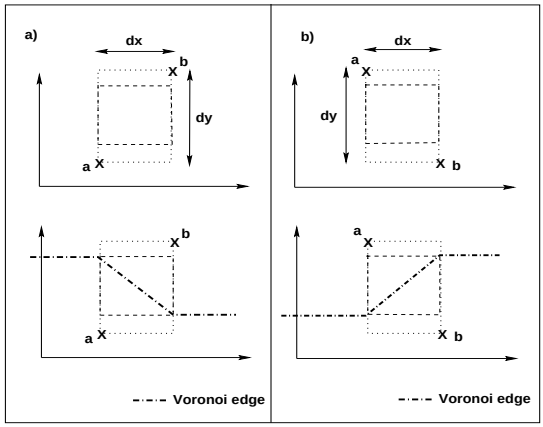


Figure 2.11: Voronoi edge for $dx < dy$

Let `VP` be a point from the Voronoi edge between points `a` and `b`. `VP` is located at the upper left corner of the dashed square in Figure 2.12.a. Let the vertical distance between `a` and `VP` be y. We know that the manhattan distance from `b` to `VP` is $dx + (dy - y)$. If `VP` indeed belongs to the Voronoi edge, its manhattan distance to both competing endpoints `a` and `b` is the same, therefore

$$\begin{aligned} y &= dx + (dy - y) \\ 2y &= dx + dy \\ y &= \frac{(dx + dy)}{2} \end{aligned} \tag{2.5}$$

Equation 2.5 provides the exact y-coordinate of `VP` when added to that of point `a`. Notice also that, as we mentioned before, the position of the voronoi point only depends on the relationship between dx and dy.

We choose now the lower right corner of the square as `VP` (Figure 2.12.b). The 2D manhattan distance from point `a` to it is $dx + y$, while the distance from `b` to `VP` is $dy - y$. Again, we equal both distances and obtain as a result Equation 2.6, which provides the y-coordinate of `VP` once added to the y-coordinate of point `a`.

$$\begin{aligned} dx + y &= dy - y \\ 2y &= dy - dx \\ y &= \frac{(dy - dx)}{2} \end{aligned} \tag{2.6}$$

The complete Voronoi edge is shown in Figure 2.12.c. The portions of the edge that extend from the corners of the square, (where the `VP` was computed) are the ones that may (or not) intersect a separator S. For a vertical separator, `pint` is obtained by projecting `VP` onto S as in Figure 2.12.c. For a horizontal separator `pint` is NULL and point `a` is kept as the closest point to S.

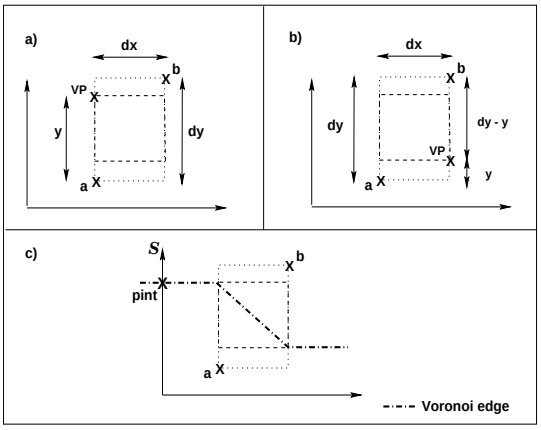


Figure 2.12: Computing step by step the Voronoi edge when $dx < dy$

- If $dx > dy$, Figures 2.8.e and Figures 2.8.f belong to this category. They have to be submitted to a squaring process, similar to the one described for the previous group. The Voronoi edges for this subcases are shown in Figure 2.13.a and Figure 2.13.b respectively.

  In order to explain the Voronoi edge computation, we use the case in Figure 2.8.e. Let the upper right corner of the dashed square in Figure 2.14.a be `VP` Let x be the horizontal distance between point `b` and `VP`.

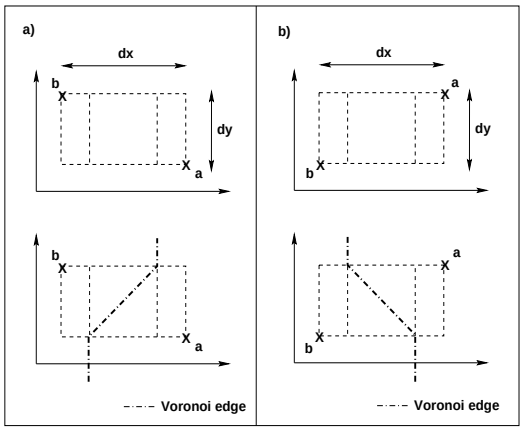


Figure 2.13: Voronoi edge for $dx > dy$

If VP is part of the Voronoi edge, its manhattan distance to each endpoint, a and b must be the same. Equation 2.7 provides the x-coordinate of VP when added to the x-coordinate of b.

$$\begin{aligned} d(\mathtt{a},\mathtt{VP}) &= d(\mathtt{b},\mathtt{VP}) \\ (dx - x) + dy &= x \\ dx + dy &= 2x \\ x &= \frac{(dx + dy)}{2} \end{aligned} \tag{2.7}$$

Now, let the lower left corner of the square (Figure 2.14.b) be VP. Equaling its manhattan distance to the endpoints we obtain Equation 2.8, shown below. The equation provides the x-coordinate of VP when added to that of b.

$$\begin{aligned} d(\mathtt{a},\mathtt{VP}) &= d(\mathtt{b},\mathtt{VP}) \\ (dx - x) &= dy + x \\ dx - dy &= 2x \\ x &= \frac{(dx - dy)}{2} \end{aligned} \tag{2.8}$$

The complete Voronoi edge is shown in Figure 2.14.c. For a vertical sweep, pint is NULL, however, it is not the case for a horizontal sweep. The other cases falling in this category are solved likewise, exploiting their geometrical similarities.

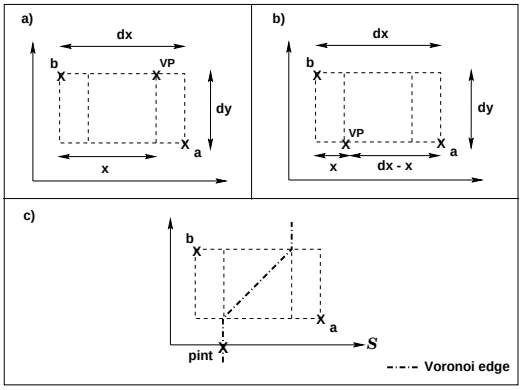


Figure 2.14: Computing step by step the Voronoi edge when $dx > dy$

2.3.3 Algorithm `Build_2D_EODM`

The algorithm that constructs the EODM is called `Build_2D_EODM()` and it is given below. Routine `Create_separators()` generates the data structure `Separators_tree`. Separators are created only once and they are linked to their adjacent white nodes. Correspondingly, each white node points to its adjacent line separators. Next, for each white node in the octree, the closest obstacle function $o_p(s)$ is computed around the boundary edges. Routine `Fill_closest_obs_EODM()` performs this process. Whenever it finds a separator S, it calls `Obtain_distance_values_for_S()` and `Copy_intervals_to_white_nodes()` to pass the distance information to all the white nodes adjacent to S.

```
Build_2D_EODM( Octree, EODM )
Input :  Octree
Output :  EODM
Separator Separators_tree
1.Create_separators( Octree, Separators_tree )
2.Fill_closest_obs_EODM( Octree->root, Separators_tree, EODM )

Fill_closest_obs_EODM( Octree_node, Separators_tree, EODM )
Input:  Octree_node, Separators_tree
Output:  EODM
Separator S
1.  For each S ∈ Separators_tree
2.      For (sweep_dir = north to sweep_dir = west)
3.          Obtain_distance_values_for_S( S, Octree_node, sweep_dir )
4.          Copy_intervals_to_white_nodes( S, Octree_node, EODM )
```

The function `Obtain_distance_values_for_S()` uses `obtain_black_nodes()` and procedure `obtain_endpoints()` to generate the list of vertices that belong to the visible[4] black nodes from S. Procedure `sweep_line()` is then called to compute $o_p(s)$.

```
Obtain_distance_values_for_S( S, Octree_node, sweep_dir )
Input:  Octree_node, S
Output:  A line separator S containing the distance intervals
1.  obtain_black_nodes( Octree_node, S, sweep_dir, Black_Nodes_List)
2.  obtain_endpoints(Black_Nodes_List, sweep_dir, Black_Endpoints)
3.  sweep_line( Sep_node, Black_Endpoints, sweep_dir )
```

[4]x-visible or y-visible depending on the direction in which the sweep is be performed

Chapter 3

Extended Octree Distance Map (EODM): 3D Case

In the previous chapter we introduced the Extended Octree Distance Map (EODM) and proposed it as an efficient collision detection scheme for static 2D environments. EODM captures the distance from a free portion of space to its closest obstacles, modeling the distance as a continuous function along such portion's border: $d_p(s)$. This section describes the extension to the three dimensional case.

To describe the building process of our approach we will consider a white node W and one of its faces F_i as an example. Let $\mathcal{R}$ denote a 3D Manhattan robot, a tetrahedron of radius r. The locus of $\mathcal{R}$ is determined by its center, situated at point L. Let $\mathcal{O}$ denote the set of obstacles of the environment. Furthermore, let $\mathcal{O}_i$ denote a single obstacle, where the subscript i indicates the obstacle number. The objective of our approach is to determine if $\mathcal{R}$, when positioned at L, is in collision with any of the obstacles $\mathcal{O}_i \in \mathcal{O}$. Both, L and r can vary from query to query.

In a quadtree, EODM keeps the distance values to the obstacles of the environment along the boundary edges of a white node W creating the distance function $d_p(s)$. For the 3D EODM the equivalent $d_p(s_1, s_2)$, a two-dimensional function (surface), is defined over the boundary faces of W. If we could measure the distance to the corresponding closest obstacle for each one of the surface points lying on the faces of W, (see Figure 3.1 below), and plot the values of $d_p(s_1, s_2)$, the result would be a "bubble" that encloses W.

Let the infinite plane in which a given face F_i of a white octree node is embedded be $\mathcal{F}_i$.

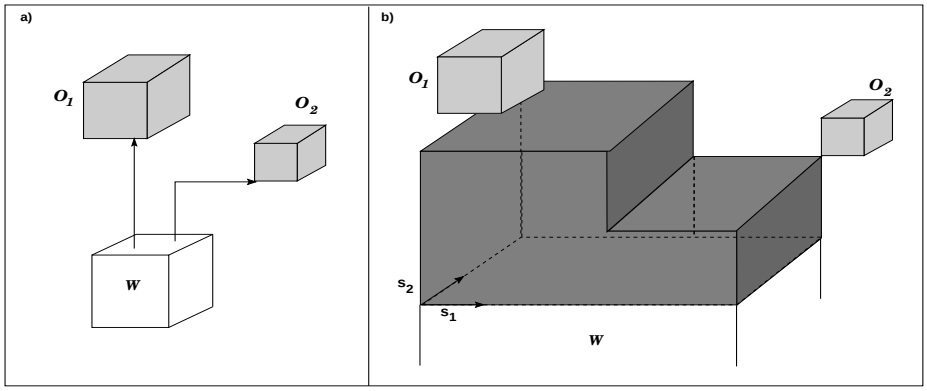


Figure 3.1: a)Closest obstacles to the top face of W b)$d_p(s_1, s_2)$ for the top face of W

We show that indeed, it is not necessary to store the distances for all the surface points of each face F_i, but only intervals along certain lines (separators) produced by the projection of the 3D obstacles over each $\mathcal{F}_i$. An interval groups the portion of a line separator for which an obstacle projection, thus its corresponding 3D obstacle, is the closest.

In order to compute the distance intervals we consider the intersection of the Voronoi Diagram of the 3D obstacles with the $\mathcal{F}_i$ that holds the obstacle projections. Such intersection generates planar Voronoi regions in $\mathcal{F}_i$. We do not explicitly compute a Voronoi region, but the intersections between its edges and the line separators using a modified sweep-line process. The sweep and the Voronoi edge computations are described in subsections 3.3.2 and 3.3.3 respectively.

It is important to remark that the way in which we proceed and the advantages that our approach possesses, are due to the cuboidal geometry of the 3D obstacles. Once projected, the projections are rectangular. This allows us to deal with projections that are parallel to the coordinate axis defined over each projection face. Had the obstacles another shape or were rotated, this feature does not hold and hence our scheme will need to be extended/modified.

The 3D EODM benefits from the qualities of a fast collision test by scaling down the problem one dimension at each phase, first from 3D to a plane by the projection of the obstacles, and then by capturing the proximity information using the line separators. This is not only reflected during collision detection time but also in the way 3D EODM is built. It is true that 3D EODM carries a considerable overhead, however, the simplicity plus the speed of the test demonstrate its advantages.

3.1 Fundamentals

The faces of an octree white node have a direction associated: NORTH, EAST, SOUTH, WEST, FRONT and BACK (shown in Figure 3.2).

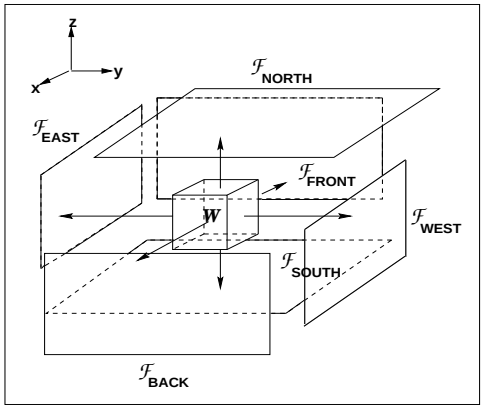


Figure 3.2: Octree white node and adjacent infinite planes

Let n̂ be the outward normal to $\mathcal{F}_i$. Let P be a point $\in \mathcal{O}_i$. We say that P is n̂-visible from $\mathcal{F}_i$ if $\exists$ a point Q $\in \mathcal{F}_i$ such that $\overline{PQ}$, the segment parallel to n̂, does not intersect any other obstacle or intersects only $\mathcal{O}_i$ exactly at P and no other point. Each $\mathcal{F}_i$ divides the space in which the robot $\mathcal{R}$ is located from the space in which the n̂-visible obstacles reside. For example, in Figure 3.3, $\mathcal{F}_{WEST}$ separates $\mathcal{R}$ from $\mathcal{O}_{WEST}$, the set of obstacles that are n̂-visible from it.

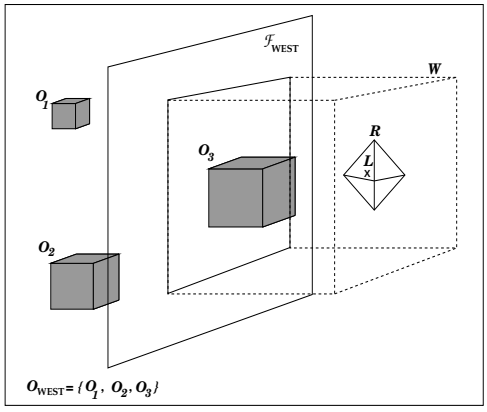


Figure 3.3: $\mathcal{F}_{WEST}$ separates $\mathcal{R}$ from $\mathcal{O}_{WEST}$

Let L, the center of $\mathcal{R}$, lie inside an octree white node W. Let L_π be the projection of

L onto the face F_{WEST} of W. The 3D manhattan distance between L and L_π is denoted by $d(L,\, L_\pi) = m$, (in Figure 3.4).

An obstacle $\mathcal{O}_i$, (a black node in the octree), that is n̂-visible from $\mathcal{F}_{WEST}$, when projected on $\mathcal{F}_{WEST}$, creates a black square, as shown in Figure 3.4. This projection is denoted as $\mathcal{O}_{i\pi}$ and we will refer to it as a black Nodal Projection. Each $\mathcal{O}_{i\pi}$ stores the distance from its corresponding 3D obstacle $\mathcal{O}_i$ to $\mathcal{F}_i$, denoted as $d(\mathcal{O},\, \mathcal{F}_{WEST}) = y$ in Figure 3.4.

Let A be a closest obstacle point to L amongst all the obstacles $\in \mathcal{O}$. In Figure 3.4, point A (on obstacle $\mathcal{O}_1$) is n̂-visible from $\mathcal{F}_{WEST}$. In fact A can belong to an n̂-visible obstacle located in any of the outward directions of the faces of W. Let the projection of A over $\mathcal{F}_{WEST}$ be A_π. A being the closest obstacle point to the robot implies that no other obstacle point exists strictly within the L_1 sphere centered at L, shown in gray in the same figure. Its radius has a length of $d(L, A) = d(L,\, L_\pi) + d(L_\pi,\, \mathrm{A}_\pi) + d(\mathrm{A}_\pi,\, \mathrm{A})$.

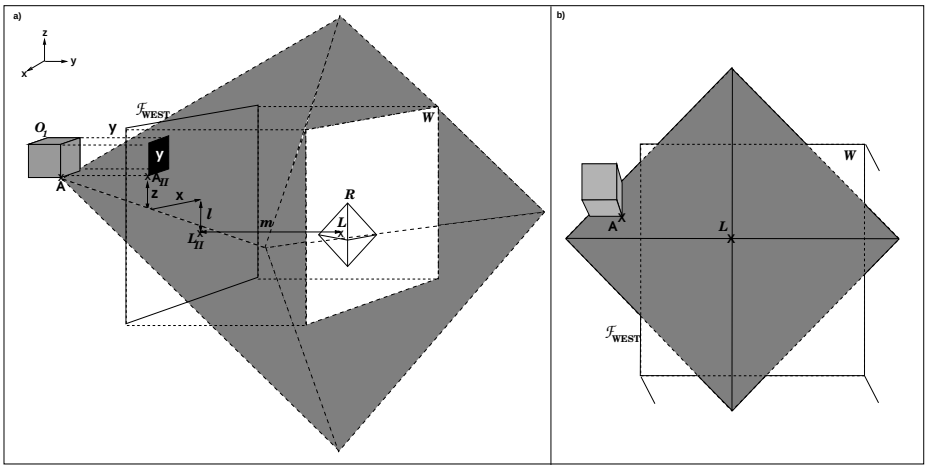


Figure 3.4: a) L_1 sphere centered at L: no other obstacle lies inside b) top view

A shortest L_1 3D path, (since it is monotone in all coordinates), between L and A passes through the following points: L, L_π, A_π and A. The length of this path is given by:

$$d_0(L) = d(L, A) = d(L, L_\pi) + d(L_\pi, A) = m + l + x + z + y \tag{3.1}$$

Equation 3.1 computes the 3D distance between any point in the face (here L_π) and its closest 3D obstacle. There may be many shortest paths connecting L with A, however, we will use the specific one illustrated in Figure 3.5. The nature of this path is characterized as

follows: $d(L, L_\pi)$ is the straight line between L and L_π and is entirely inside W, therefore it is free. $d(L_\pi, A)$ is the distance from the robot projection to a closest point A that belongs to an obstacle. Computing $d(L_\pi, A)$, in Equation 3.1, efficiently is the key to our approach. We compute it by projecting the n̂-visible obstacles of each $\mathcal{F}_i$ onto it. The projections induce a rectangular partition (grid) on the infinite planes by extending their boundary line segments (see Figure 3.6).

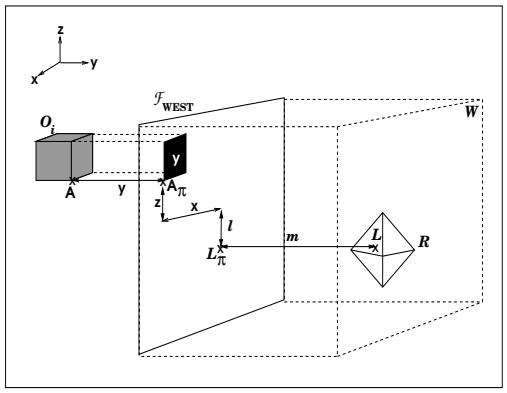


Figure 3.5: Distance between a robot in W and its closest obstacle

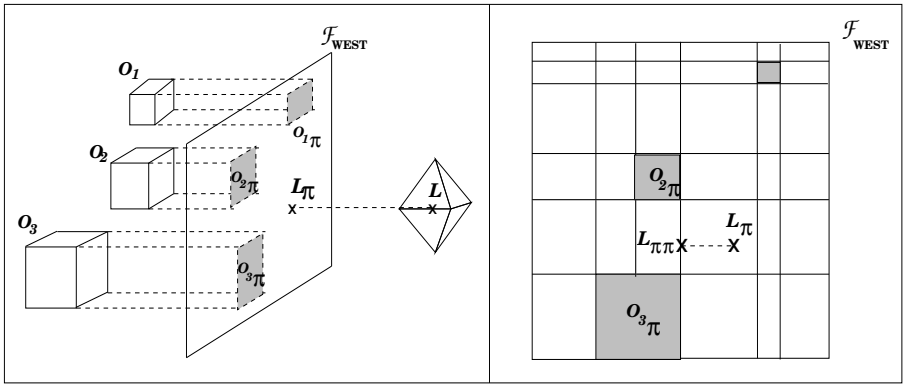


Figure 3.6: Rectangular partition generated by the projection of the obstacles

Now, we decompose $d(L_\pi, A)$ from Equation 3.1 into

$$d(L_\pi, A) = d(L_\pi, L_{\pi\pi}) + d(L_{\pi\pi}, A) \tag{3.2}$$

where $d(L_\pi, L_{\pi\pi})$ is the manhattan distance between L_π and its projection point $L_{\pi\pi}$ onto a line of the grid, and $d(L_{\pi\pi}, A)$ is the manhattan distance from the projection point

on the grid, $L_{\pi\pi}$, to the closest obstacle point A. The term $d(L_{\pi\pi}, A)$ is the proximity information that we store as a set of intervals along the lines of the grid. Each interval is associated with the 3D obstacle that is closest to that portion of the line.

Figure 3.7 shows a line of the grid and the intervals produced by the obstacles of Figure 3.6. Interval $\mathtt{I}_2$ is associated with obstacle $\mathcal{O}_2$, and interval $\mathtt{I}_3$ has been associated with $\mathcal{O}_3$. At collision detection time, when L_π is projected onto a line of the grid, its projection $L_{\pi\pi}$ falls in at least one of the intervals computed for that particular line. Accessing the interval where $L_{\pi\pi}$ falls leads us right away to the closest 3D obstacle and the computation of the term $d(L_{\pi\pi}, A)$ in Equation 3.2 is then carried out.

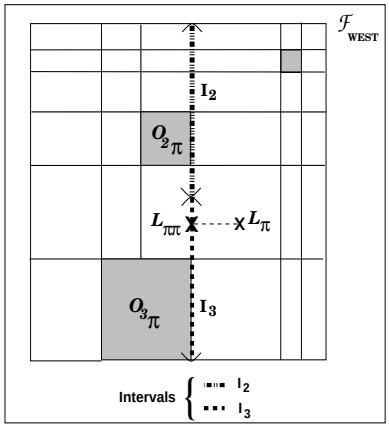


Figure 3.7: Distance intervals stored along a line of the grid

In equation 3.1, $d_O(L)$ provides a bound for the radius r of the robot. If $r \geq d_O(L)$ the robot collides with an obstacle $\in O$, otherwise the robot is collision free. Collision detection requires first to find the node W in which the center of the robot is located. Another search must be performed to find the region in the grid of each adjacent $\mathcal{F}_i$ of W in which the projection of the center of the robot falls. Once this stage is passed, collision detection with a 3D EODM becomes a simple test that can be performed in constant time. The details are discussed in section 3.2.

The obstacles' projections divide $\mathcal{F}_i$ into 2D rectangular cells, which are colored either black or white. A black cell (called Nodal Projection) represents pseudo-obstacles, whereas a white cell represents a portion of the plane for which there are no 3D obstacles lying directly above it. Let the cell in which L_π falls be c_i.

Any cell divides the plane of an $\mathcal{F}_i$ into four overlapping regions (north, south, east

and west) that extend to the outward directions around the boundary of the cell (see Figure 3.8.a). A black Nodal Projection requires an extra region denoted top because a 3D obstacle lies directly "above" it. These regions are obtained by extending to infinity the edges of the Nodal Projections. The stretched line segments are called "line-separators". Figure 3.8.b shows the regions created by each kind of cell. For Figure 3.6, the projection $\mathcal{O}_{i\pi}$ of the closest n̂-visible 3D obstacle to L_π from $\mathcal{F}_{WEST}$ must lie in either at least one of the four regions around c_i or overlap it completely. The same principle applies to all the infinite planes adjacent to W.

Each line-separator that bounds the cell divides the plane into two half planes. Similar to our 2D EODM, one half plane corresponds to the outward normal direction, the other half plane contains L_π. Let $\mathcal{O}_{\pi WEST_{(north)}}$ represent the set of Nodal Projections on $\mathcal{F}_{WEST}$ (of all the n̂-visible 3D obstacles from $\mathcal{F}_{WEST}$), that lie on the north outward normal of a line separator S of cell c_i.

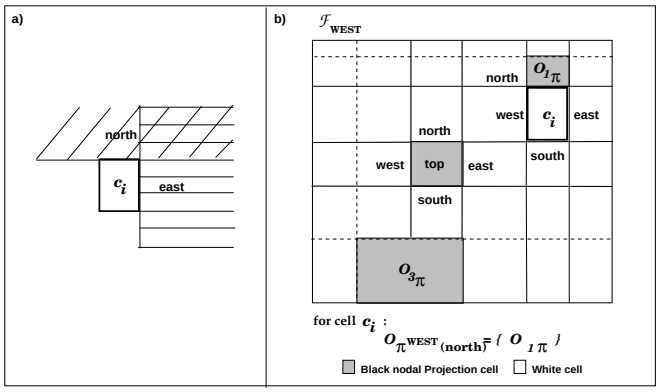


Figure 3.8: a) North and West regions of a cell b) Regions for black and white cells of $\mathcal{F}_{WEST}$

Up to this point, we have provided the characteristics of the $\mathcal{F}_i$'s. Now we will characterize the shortest path between L_π and the closest obstacle point. There are two cases, either L_π overlaps an $\mathcal{O}_{i\pi}$ when projected onto an $\mathcal{F}_i$ or it does not. The characteristics of both cases will be discussed in the rest of this section.

3.1.1 L_π falls in a white cell

Consider all the n̂-visible 3D obstacles from the WEST face of a white cube shown in Figure 3.9. Focus only on the group of projections contained in $\mathcal{O}_{\pi WEST(north)}$. Let point A be an obstacle point (from $\mathcal{O}_i \in \mathcal{O}_{WEST(north)}$). Let A_π be the projection of A onto

$\mathcal{F}_{WEST}$. Moreover, let L_π be contained in a white cell and let $L_{\pi\pi}$ be the projection of L_π on the (north) line separator S which separates A_π from L_π.

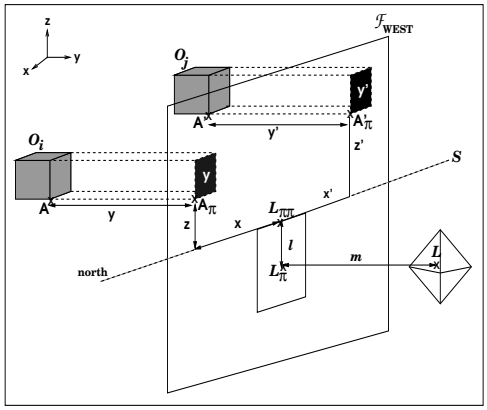


Figure 3.9: Proof for Lemma 2 with 3D L_1 metric

Lemma 2 If point A is a closest obstacle point to $L_{\pi\pi}$, then it must be the closest obstacle point (from $\mathcal{O}_{WEST_{(north)}}$) to L_π, thus, to the robot.

Proof The proof is by contradiction. Assume there $\exists$ another point A' (from the same or another obstacle $\mathcal{O}_j \in \mathcal{O}_{WEST(north)}$) strictly closer (than A) to L_π. Figure 3.9 shows the corresponding example. Notice that the projections of the obstacles over $\mathcal{F}_{WEST}$ lie above (outward direction of) the infinite line, (shown in dashed), passing through the north boundary of the cell that contains L_π.

The monotonic path between L and A' must be included within the L_1 sphere with center at L and radius $d(L, A)$. Additionally, we have $d(L_\pi, A') = x' + y' + z' + l$ (a monotonic path, hence the shortest). Now, by assumption

$$d(A', L_\pi) < d(A, L_\pi)$$

But, we know

$d(A, L_\pi) = x + y + z + l$ and $d(A', L_\pi) = x' + y' + z' + l$

$$\Rightarrow x' + y' + z' + l < x + y + z + l$$

$$\Leftrightarrow x' + y' + z' < x + y + z$$

$\Rightarrow A'$ is closer to $L_{\pi\pi}$ than A, a contradiction. $\diamond$

We extend Lemma 2 further by projecting L_π onto each boundary of c_i. L_π generates an $L_{\pi\pi(edge\_dir)}$ corresponding to the direction $edge\_dir \in$ {north, east, south, west} in which the projection occurs.

Now, let

$$\mathcal{O}_{c_i} = \{\mathcal{O}_{\pi WEST(north)} \cup \mathcal{O}_{\pi WEST(east)} \cup \mathcal{O}_{\pi WEST(south)} \cup \mathcal{O}_{\pi WEST(west)}\}$$

represent the set of obstacle Nodal Projections on $\mathcal{F}_{WEST}$, surounding c_i. All the projections in $\mathcal{O}_{c_i}$ fall in at least one of the regions generated by the cell. Applying Lemma 2 to each $L_{\pi\pi(edge\_dir)}$ will return the closest obstacle projection point in each edge direction ($\mathrm{A}_{WEST_{closest\_from\_edge}}$), where each

$$\mathrm{A}_{WEST_{closest\_from\_edge}} \in \mathcal{O}_{c_i}$$

Theorem 1 The distance between L and $\mathrm{A}_{WEST_{closest\_from\_edge}}$ is the length of the shortest monotonic path between the robot and the closest obstacle. Clearly,

$$A_{WEST_{closest\_from\_edge}} = \min_{edge\_dir} \{\ d(L_{\pi\pi_{(edge\_dir)}}, \mathcal{O}_{\pi WEST(edge\_dir)})\ \} \tag{3.3}$$

where $edge\_dir = \{\ north \mid east \mid south \mid west\ \}$

3.1.2 L_π overlaps a Nodal Projection

When L_π falls inside of a black Nodal Projection it is necessary to consider, in addition, the shortest path between L_π and the 3D obstacle lying directly over it in the region denoted as top. Let points A and A$'$ be obstacle points (from the same or different obstacles) visible from $\mathcal{F}_{WEST}$. There are two cases. In the first one, both, A and A$'$ belong to a region of c_i which is not the top one (for example north). In the second case A falls in the top region. We present now the two subcases of Lemma 3 applied on $\mathcal{F}_{WEST}$.

Lemma 3.a If point A is a closest obstacle point to $L_{\pi\pi}$, then it must be the closest obstacle point (from $\mathcal{O}_{WEST_{(north)}}$) to L_π, thus, to the robot.

Proof The proof is exactly similar to that of Lemma 2 applied for each edge of the cell that contains L_π.

Lemma 3.b, shown below, is also applied over $\mathcal{F}_{WEST}$, (see Figure 3.10 for an example). Notice that the closest obstacle projection point to the robot, (point A_π), is exactly L_π.

Lemma 3.b If point A is a closest obstacle point $\in \mathcal{O}_i$ (amongst obstacles $\in \mathcal{O}_{WEST}$), to the projection point L_π, then it must be a closest obstacle point to the robot.

Proof We present the following proof by contradiction. Assume there $\exists$ another point $A^{'}$ from the same or another obstacle $\mathcal{O}_j$. Let A $\in \mathcal{O}_{WEST(top)}$ and let $A^{'}$ be in the same or another region, lying strictly closer (than A) to L, and $d(L, A^{'}) = x^{'} + y^{'} + z^{'} + l + m$ (a monotonic path, hence the shortest).

$$\Rightarrow d(L, A^{'}) < d(L, A)$$

But, we know $\quad d(L, A) = y + m \quad$ and $\quad d(L, A^{'}) = x' + y' + z' + l + m$

$$\Leftrightarrow x' + y' + z' + l + m < y + m$$

$$\Leftrightarrow x' + y' + z' + l < y$$

$\Rightarrow A^{'}$ is closer to L_π than A, a contradiction. $\diamond$

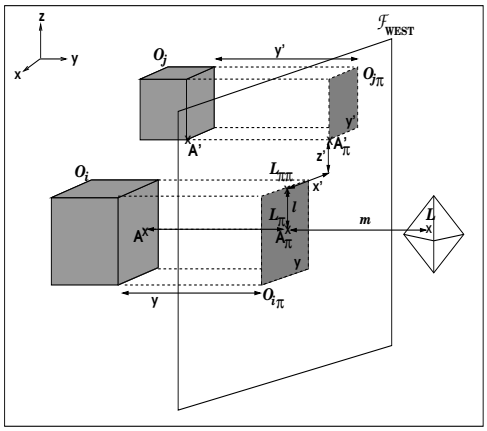


Figure 3.10: Proof for Lemma 3.b with 3D L_1 metric

To characterize the closest obstacle point from $\mathcal{F}_{WEST}$ for Lemma 3, we extend Equation 3.3 as:

$$A_{WEST_{closest\_from\_edge}} = \min_{edge\_dir} \{ d(L_{\pi\pi_{(edge\_dir)}}, \mathcal{O}_{\pi WEST(edge\_dir)}) \} \tag{3.4}$$

where $edge\_dir = \{ north \mid east \mid south \mid west \mid top \}$

Using Lemmas 2 and 3 we have shown that the closest obstacle to the robot, once projected, is the closest Nodal Projection to L_π, no matter if the robot projection falls in a black or a white cell. However, they apply only amongst $\mathcal{O}_{WEST}$. We show next that our proof extends to all obstacles.

3.1.3 Closest Obstacle Point Characterization

Let $face\_dir$ denote the directions (NORTH, EAST, SOUTH, WEST, FRONT, BACK) in which L, the center of the robot, is projected over the faces of W. Similarly, $L_{\pi face\_dir}$ denotes the projection point of L in each face_dir. Let $\mathrm{A}_{face\_dir_{closest\_from\_edge}}$ be the closest obstacle point to each $L_{\pi face\_dir}$ as defined in Theorem 1 and its extension in Equation 3.4.

Theorem 2 The closest obstacle point $\mathrm{A}_{closest}$ to L is given by

$$A_{closest} = \min_{face\_dir} \{ d(\ L, A_{closest\_from_{face\_dir}}\ ) \} \tag{3.5}$$

where

$$A_{closest\_from_{face\_dir}} = \min_{closest\_from\_edge} \{ d(\ L_{\pi_{face\_dir}}, A_{face\_dir_{closest\_from\_edge}}\ ) \} \tag{3.6}$$

and $face\_dir = \{NORTH|EAST|SOUTH|WEST|FRONT|BACK\}$

Theorems 1 and 2 guarantee that the computation of the closest 3D obstacle to the robot is performed taking in account all the 3D obstacles of the environment. Because we capture the proximity information around the boundaries of the cells in the grid, the internal part of the Nodal Projections is discarded during the distance computation (see sections 3.3.1 and 3.3.2).

In the following subsection, both, the description and the algorithm for our collision detection scheme with a 3D EODM will be provided.

3.2 3D Collision Detection

In order to explain how collision detection is performed for a 3D EODM, consider Figure 3.11, in which the manhattan 3D sphere represents our robot $\mathcal{R}$, positioned at location L.

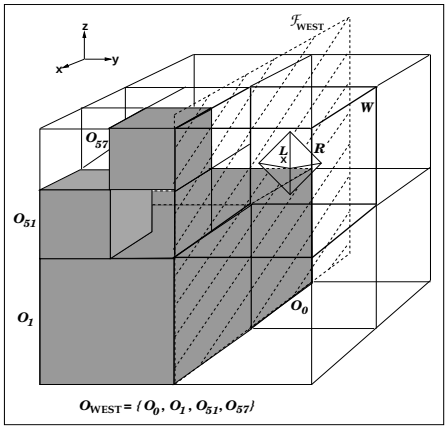


Figure 3.11: Robot lying in an octree

The collision detection test with the 3D EODM is quite simple once the coarsest resolution node W, in which the center of $\mathcal{R}$ lies, has been found. In the pseudo-code, shown below, procedure `find_white_node()` reports such a node in $O(logN)$ for an octree with N leaf nodes. `Min_dist` denotes the 3D manhattan distance between the 3D `Closest_Obstacle` and $\mathcal{R}$. Originally, `Min_dist` is initialized to an arbitrary large value.

If the node W is black, a collision is returned immediately. Otherwise, the center of $\mathcal{R}$ is projected to all the boundary faces that compose W, producing the corresponding projection points denoted by $L_{\pi face\_dir}$. In our sample figure, the center of the robot is projected over face $\mathcal{F}_{WEST}$ and generates point $L_{\pi WEST}$. Let `dist_face` denote the 3D manhattan distance between the center of the robot and its projection $L_{\pi face\_dir}$, (first term of Equation 3.1).

We mentioned that each F_i of W has associated an infinite plane $\mathcal{F}_i$. For our computation purposes, we limit $\mathcal{F}_i$ to the workspace boundary. Each face plane $\mathcal{F}_i$ is examined in turn by the procedure `Detect_Collision_in_face()` (line 9 of the below pseudo-code). `Detect_Collision_in_face()` not only returns the COLLISION/FREE `Status` of the collision test for a single face, but also the `Closest_Obstacle` to L_π, (thus returning the closest 3D obstacle to the robot by Lemma 2 and Lemma 3), and its manhattan 3D distance

(`Min_dist`).

The pseudo-code of our three dimensional collision test is stated as follows:

```
Detect_3D_Collision( EODM, Robot, Closest_obst, Min_dist )
Input :  EODM, Robot
Output:  COLLISION/FREE status, Closest_obst, Min_dist

1.  W = find_node( EODM, Robot->center )
2.  if ( COLOR(W ) == BLACK )
3.      Closest_obstacle = W
4.      Min_dist = 0
5.      return COLLISION
6.  For each face F_i of W
7.      L_π = project_robot( Robot, F_i )
8.      dist_face = distance_to_face ( Robot, L_π )
9.      Status = Detect_Collision_in_face( F_i, L_π, dist_face, Closest_obst, Min_dist )
10.     if (Status)
11.               return COLLISION
12. return FREE
```

For the moment, assume that each face plane $\mathcal{F}_i$ of W has already captured the proximity information $d(L_\pi, A)$ from Equation 3.1.

Given a particular face direction, let `dist_to_face` be $d(L, L_\pi)$, the L_1 distance between L_π and the center of the robot. L_π must fall in at least one of the cells of $\mathcal{F}_i$. Let such cell be c_i. For every c_i, irrespective of its color, all its edges (line separators), are tested for collision using the function `Detect_collision_for_cell()`, (line 21 below pseudo-code). The robot center projection L_π is projected again over each edge that bounds c_i, originating the corresponding point $L_{\pi\pi}$. If c_i is a Nodal Projection, then an extra collision test with the 3D obstacle that generated it is required (lines 14-18 of the pseudo-code shown below).

The cell c_i is examined in each edge direction (north, east, south, west), as in our 2D EODM model. Procedure `Detect_collision_for_cell()` receives the edge `Edge` that bounds c_i and the projection $L_{\pi\pi}$ of the robot center onto that particular edge. The L_1 distance between L_π and `Edge`, $d(L_\pi, L_{\pi\pi})$, is denoted by `dist_edge` (first term in Equation 3.2).

```
Detect_Collision_in_face( F_i, Robot, L_π, dist_face, Closest_obst, Min_dist )
Input :  F_i, Robot, L_π, dist_face
Output:  Status(COLLISION / FREE), Closest_obst, Min_dist

13. c_i = find_cell( F_i, L_π->center )
14. if ( COLOR( c_i ) == BLACK )
15.     Min_dist = dist_to_obstacle( Robot->center, c_i ->3d_obst )
16.     Closest_obst = c_i ->3d_obst
17.     if ( Min_dist < Robot->radius )
18.             return COLLISION
19. for (Edge = NORTH to WEST )
20.     L_ππ= project_robot( Edge, L_π)
21.     Status = Detect_collision_for_cell( Edge, Robot, L_ππ,
                dist_face, Closest_obst, Min_dist )
22.     if ( Status )
23.             return COLLISION
24. return FREE
```

Because `Edge` is a line separator, it stores a sequence of intervals. An interval is associated to the Nodal Projection that is closest to that particular portion of `Edge` on its outward direction. Let `I` be the interval along `Edge` in which $L_{\pi\pi}$ is contained. `I` points to its closest 3D obstacle. The distance between them is `d_interval_to_obstacle`, which provides the last component for computing $d_O(L)$ in Equation 3.1, (line 27 of the pseudo-code shown below). $d_O(L)$ is represented by `dist_to_clst_obstacle`. In practice, our implementation computes the distance between `I` and its associated 3D obstacle at runtime, a step that requires only simple algebraic operations. `dist_to_clst_obstacle` is the bound against which the radius of the robot is compared in order to find a collision (lines 28-32).

```
Detect_collision_for_cell( Edge, Robot, L_ππ, dist_face, Closest_obst, Min_dist )
Input :  Edge, Robot, L_ππ, dist_face
Output :  Status, Closest_obst, Min_dist

25. I = get_interval( L_ππ, Edge)
26. dist_edge = dist_to_edge( L_ππ, Edge )
27. dist_to_clst_obstacle = dist_face + dist_edge + I->d_interval_to_obstacle
28. if ( dist_to_clst_obstacle <= Robot->radius )
29.     if ( dist_to_clst_obstacle < Min_dist )
30.             Min_dist = dist_to_clst_obstacle
31.             Closest_obst = I->3d_obst
32.     return COLLISION
```

The next section will discuss the creation of the 3D EODM.

3.3 3D EODM Creation

As in the two dimensional problem, there are two possibilities to construct the EODM, a global and a local approach. The local approach captures the proximity to the obstacles only within the neighborhood of the white cube W. Only the 3D black nodes within the parent cell of W are projected over its boundary faces. The global approach considers all the obstacles of the environment at once, which means that all the $\hat{n}$-visible obstacles from each boundary face of W will be projected over it. The global approach requires some extra memory to maintain the connectivity between the white nodes of the EODM and the adjacent face separators, which make it slower to construct when compared to the local approach. We have chosen to work with the global approach mainly because the extra computations required by the local approach during collision detection time (the distance information must be retrieved and tested against the robot at each level of the tree until W is found), will make it somewhat slower.

The infinite plane separators adjacent to the faces of the white nodes are obtained each time the EODM is split. A face separator divides the free space contained inside of a white node from the 3D $\hat{n}$-visible obstacles. Three kinds of face separators are considered, (shown in Figure 3.12). A face separator contains the proximity information to the 3D obstacles lying to its left and to its right (above and below for the horizontal separators), for this reason each face separator contains two grids.

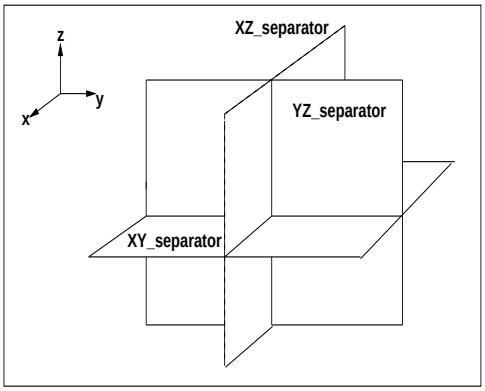


Figure 3.12: Face separators in 3D

The data structure of an EODM stores the dimensions of the environment and a pointer to the root node. The number of child nodes of an EODM, `num_ch`, is a function of 2^d, where

d is the dimension of the workspace of the robot. Each white node of our EODM stores a pointer to its adjacent face separators. `Make_3D_EODM()` is the procedure responsible for constructing the EODM, creating the data structures for the face separators, connecting them with the white nodes and filling the distance information. The corresponding pseudo-code is shown at the end of this subsection.

All the children of a 3D EODM are examined in turn by the procedure `Connect_3D_EODM()` (line 7 of the below pseudo-code). If a white child is found, all its boundary faces are linked to its adjacent face separators by the procedure `link_white_node()`. Pseudo-codes for the functions `Connect_3D_EODM()` and `link_white_node()`, can be found in Appendix A.

Separate tree structures have been considered to organize the face separators. Each face separator is inserted by procedure `insert_separator()` in a tree hierarchy during the EODM split process (see Appendix A). The hierarchies of face separators are aligned with respect to each coordinate plane in 3D space and are identified by `XY_separator`, `XZ_separator` and `YZ_separator`. The generic name used for plane separators is `sep_tree`.

Procedure `fill_3D_separators()` obtains the distance information for each face separator (lines 5-7 of the below pseudo-code). The construction stage of the face separators is explained in the next subsection.

```
Octree Make_3D_EODM( otree, EODM )
Input :  simple octree otree
Output :  an EODM

node_level              level
child_number            num_ch
octree_node             currnode
3D_EODM                 eodm
1.  num_ch              = power(2.0, otree->dimen)
2.  level               = 0
3.  eodm   = Connect_3D_EODM( otree->root, level, num_ch )
4.  currnode = eodm->root
5.  fill_3D_separators( eodm, currnode, level, XY_separator )
6.  fill_3D_separators( eodm, currnode, level, XZ_separator )
7.  fill_3D_separators( eodm, currnode, level, YZ_separator )
```

3.3.1 Projecting the obstacles

In order to project the n̂-visible obstacles from a face $\mathcal{F}_i$, we require a `Z_buffer` structure. For example, for an $\mathcal{F}_i$ parallel to the XY plane, the `Z_buffer` orders the n̂-visible obstacles taking in account their proximity to $\mathcal{F}_i$ using the z-dimension of the vertices of the obstacle faces that are perpendicular and that directly face $\mathcal{F}_i$.

Our approach does not store a cell of the grid *per se* but only the portion of the boundary of the cell that overlaps an adjacent line separator. We store those portions as intervals. During the projection, Nodal Projections can overlap, therefore, clipping and updating operations are performed for the intervals of the affected line separators. The resulting new intervals must be associated correctly with their originating 3D obstacles. Figure 3.13 shows some examples. We use a dashed line to represent the intervals created by the Nodal Projection of $\mathcal{O}_1$ and a solid line to represent the intervals generated by the Nodal Projection of $\mathcal{O}_2$ along a line separator S. Figure 3.13.a shows S and the interval $\mathtt{I}_2$, generated by $\mathcal{O}_2$. Even though both obstacles are projected over the plane $\mathcal{F}_i$, only obstacle $\mathcal{O}_2$ generates intervals on all the line separators because it is closest to the plane and occludes completely $\mathcal{O}_1$. Figure 3.13.b shows a partial obstacle obstruction, where each obstacle, when projected, generates an interval on S. Finally, in Figure 3.13.c $\mathcal{O}_2$ is projected first and then $\mathcal{O}_1$. When $\mathcal{O}_1$ is projected it creates separator lines, (including S), and the interval $\mathtt{I}_1$. The Nodal Projection of $\mathcal{O}_2$ overlaps the separators created by the projection of $\mathcal{O}_1$, (overlapping S too), therefore two intervals $\mathtt{I}_2$ are created and associated with $\mathcal{O}_2$.

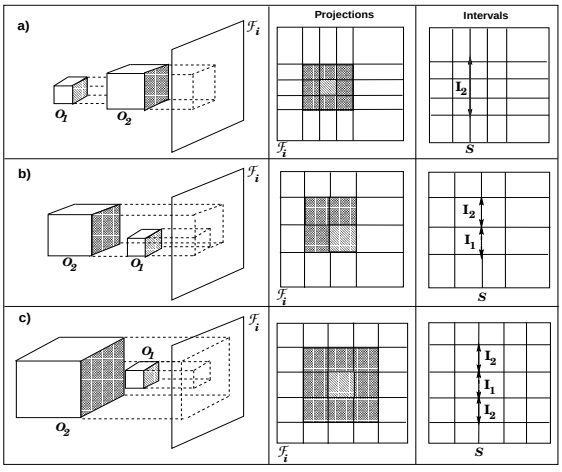


Figure 3.13: Boundaries associated with each Nodal Projection

The algorithm that computes the proximity information for the plane separators is called `fill_3D_separators()`, introduced in the previous subsection. For each face separator $\mathcal{F}_i$ in `sep_tree` we construct two grids in order to capture the distance information of the $\hat{n}$-visible 3D obstacles lying on each side of $\mathcal{F}_i$. The core of `fill_3D_separators()` is shown next.

```
fill_3D_separators( otree, currnode, level, sep_tree )
Input :  Octree otree, otree_node currnode, node_level level, Separator sep_tree
Output:  sep_tree filled with the distance intervals

1.  if (sep_tree == NULL)
2.          return
3.  Z_buffer = order_visible_obstacles( otree, sep_tree )
4.  Black_Nodal_Proj = project_obstacles( otree, sep_tree, Z_buffer )
5.  Build_grid( sep_tree, level, Black_Nodal_Proj )
6.  fill_3D_separators( otree, currnode, level+1, sep_tree->left )
7.  fill_3D_separators( otree, currnode, level+1, sep_tree->right )
```

The `Z_buffer` is constructed by `order_visible_obstacles()`. The obstacles are projected over the face separator using `project_obstacles()` starting with the farthest. The Nodal Projections are kept in the list `Black_Nodal_Proj`. Then, the grid is constructed by procedure `Build_grid()`, which applies the sweep-line operation to `Black_Nodal_Proj` in order to fill the 2D line separators. The whole process is repeated recursively for all the remaining face separators in `sep_tree`. The distance intervals computation using the sweep-line is described next.

3.3.2 Computing the intervals for the line separators

There are two kinds of line separators in a grid, vertical and horizontal. [1] The process described in this subsection applies to all the line separators of a grid. To make the contents of this subsection easier to explain, we have chosen to work with a vertical $\mathcal{F}_i$ and only with its $\hat{n}$-visible obstacles lying in the WEST direction. The sweep will be applied over a horizontal line separator using the Nodal Projections lying on its north outward direction[2],

[1] We use the terms vertical and horizontal somewhat arbitrarily. They denote separators parallel to the axes within the projection plane.

[2] Our implementation also takes into account those Nodal Projections that are cut by a line separator.

in other words, only the Nodal Projections $\in \mathcal{O}_{WEST_{(north)}}$ of a cell c_i are considered (see Figure 3.14).

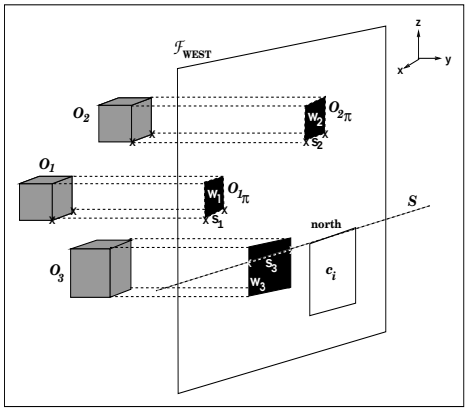


Figure 3.14: Visible obstacle projections from the north boundary of c_i

Let $\mathcal{O}_{\pi_{WEST_{(north)}}}$ be the set of black Nodal Projections generated by $\mathcal{O}_{WEST_{(north)}}$. For our example in Figure 3.14, $\mathcal{O}_{\pi_{WEST_{(north)}}} = \{ \mathcal{O}_{1\pi}, \mathcal{O}_{2\pi}, \mathcal{O}_{3\pi}\}$ = `Black_Nodal_Proj`. Let `Black_segments` represent the list of those segments of the Nodal Projections $\in \mathcal{O}_{\pi_{WEST_{(north)}}}$ that are visible[3] from the north direction of S. For the Nodal Projections in Figure 3.14 `Black_segments` $= \{s_1, s_2, s_3\}$. A segment $\in$ `Black_segments` stores the distance between its 3D obstacle and the projection face as a weight $\mathtt{w}_i$.

In theory, a projection face $\mathcal{F}_i$ extends to infinity, however, for programming purposes it is bounded by the dimensions of the workspace. For the same reason, each line separator S, which in theory is an infinite line, is also limited by the boundaries of the projection face. Each bounded $\mathcal{F}_i$ has a well defined coordinate frame. We also associate an origin point with each line separator using the face's coordinate frame as a reference.

The endpoints of each segment included in `Black_segments` are sorted in a list, (called `Ordered_endpoints`), according to their 3D L_1 distance to the origin of S. [4] The sorting is performed by procedure `sort_endpoints()`.

[3] x-visible or y-visible depending on the direction in which the sweep is performed.

[4] This distance not only includes the length of the path from an endpoint to the origin of the separator, but also the weight $\mathtt{w}_i$ of the segment that owns such endpoint.

A `sweep_line()` process is used to traverse all the endpoints in `Ordered_endpoints`. Two events stand out: the sweep-line, `L`, crosses for the first time a segment (a left endpoint is encountered) or the sweep line abandons a segment (a right endpoint is traversed). Some bookkeeping operations must be performed in each case. To keep track of all the segments that are encountered by the sweep-line at a particular time, a list of `Active_segments` is maintained. When the sweep-line crosses a segment `s`, the operation `above(s, Active_segments)` returns the active segment lying immediately above (or collinear with) segment `s` that is also intersected by the sweep-line. Additionally, procedure `below(s, Active_segments)` returns the active segment located immediately below (or collinear with) `s` that is also intersected by the sweep-line. The partial order (above, or below) in `Active_segments` is given by the y-coordinate of the endpoints of the segments for an horizontal sweep. When a vertical sweep is performed, the order is maintained by the x-coordinate of the segments' endpoints.

Let the endpoint that is being currently examined by the sweep line be `current_point`. If `current_point` is a left endpoint, its segment `s`, is inserted into `Active_segments`. If the sweep is horizontal and performed in the north direction the insertion takes into account the x-coordinate of the left endpoint of `s`. For a sweep performed in the south direction, the right endpoint of `s` is used. When a vertical sweep in the east or west direction is performed, the y-coordinate of the lower endpoint of `s` is used.

Once inserted, segment `s` has to compete with its adjacent neighbors in `Active_segments` to verify if it can generate a new interval along S, in other words, to verify if `s` is closer to S than any other segment within its neighborhood. Such neighbors are identified by `Above_active_seg` and `Below_active_seg`. In case they exist, they must have been inserted by previous iterations in `Active_segments`.

To possibly generate a new distance interval, procedure `compute_voronoi()`, (lines 7, 10 and 16 of the below pseudo-code), obtains the intersection point, `VP`, of the Voronoi edge between two given segments and S. If `VP` exists, `update_intervals()` manipulates the intervals of S according to the event code produced by `compute_voronoi()`. An event code determines when to open a new interval, close, or preserve the current interval of S. An interval is updated only if `VP` falls beyond the intersection of the sweep-line (passing through `current_point`) and S. The opposite situation implies that there exists another segment closer to S than the current segment `s` that was examined in a previous iteration. In this case, the current interval is not altered. The Voronoi edge computation and the

criteria applied to obtain the event codes for the example that will be presented in this section are discussed in section 3.3.3.

Given two segments s1 and s2, `compute_voronoi(s1, s2)` returns the following event codes:

- 1 = keeps `s1` as the closest segment to the current interval
- 2 = `s2` remains as the closest segment to the current interval
- 21 = closes the current interval, associated with `s2`, and opens a new one for `s1`
- 12 = closes the current interval, associated with `s1`, and opens a new one for `s2`.

The pseudo-code for the sweep-line algorithm is presented below:

```
sweep_line( S, Black_segments )
Input :  S, Black_segments
Output:  S modified with the intervals

1.  Ordered_endpoints = sort_endpoints( Black_segments, S->origin )
2.  do
3.      current_point = top(Ordered_endpoints)
4.      if (current_point is a left endpoint)
5.          insert( current_point->segment, Active_segments )
6.          if ( Above_active_seg = above(current_point->segment, Active_segments ) exists )
7.              VP = compute_voronoi(current_point->segment, Above_active_seg)
8.              update_intervals( S->intervals, VP )
9.          if ( Below_active_seg = below(current_point->segment, Active_segments ) exists )
10.             VP = compute_voronoi(current_point->segment, Below_active_seg)
11.             update_intervals( S->intervals, VP )

12.     if (current_point is a right endpoint)
13.         Above_active_seg = above(current_point->segment, Active_segments)
14.         Below_active_seg = below(current_point->segment, Active_segments )
15.         if ( Above_active_seg && Below_active_seg exist )
16.             VP = compute_voronoi( Above_active_seg, Below_active_seg )
17.             update_intervals( S->intervals, VP )
18.         delete( current_point->segment, Active_segments )
19.     Ordered_endpoints = Ordered_endpoints->next
20. while (Ordered_endpoints is not empty)
```

When `current_point` is a right endpoint, the current segment is no longer traversed by the sweep-line. Before it is deleted from `Active segments`, it is necessary to examine if the neighboring segments (`Above_active_seg` and `Below_active_seg`) can generate a new interval (lines 12-17 of the `sweep_line()` pseudo-code).

To demonstrate how the sweep line algorithm operates, consider the example illustrated in Figure 3.15. Let the white node where the sweep is applied be node 5 and the projection face be $\mathcal{F}_{WEST}$. The n̂-visible 3D obstacles belong to the set $\mathcal{O}_{WEST}$. Note that only those projections in $\mathcal{O}_{\pi WEST(north)}$ (lying above or intersecting S) are taken in account for the sweep. For the sample figure $\mathcal{O}_{\pi WEST(north)} = \{ \mathcal{O}_{2\pi}, \mathcal{O}_{32\pi}, \mathcal{O}_{33\pi}, \mathcal{O}_{7\pi} \}$.

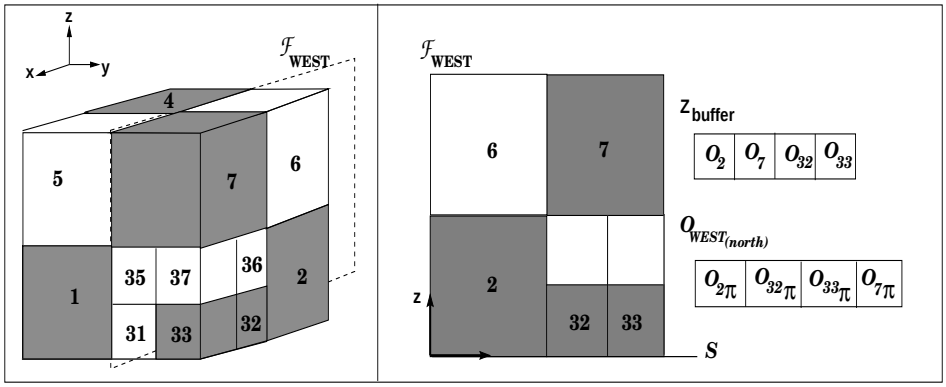
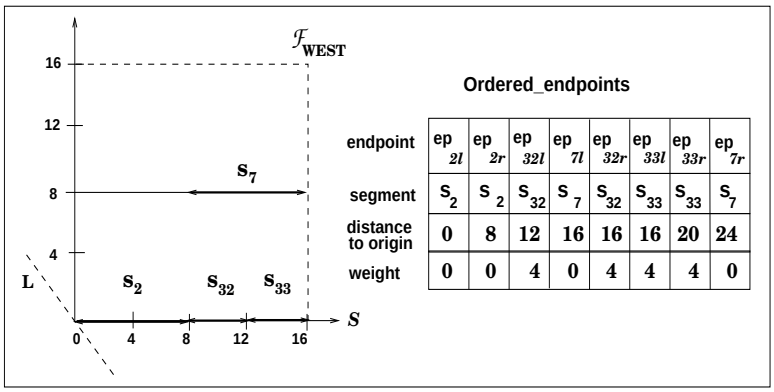


Figure 3.15: Visible obstacles from $\mathcal{F}_{WEST}$ for node 5

The notation for the endpoints of the segments in our example is $\mathtt{ep}_{(s,e)}$, where `s` indicates the segment to which the endpoint belongs and `e` indicates whether it is a left (l) or a right (r) one.

From $\mathcal{O}_{\pi WEST(north)}$, the y-visible segments from the north boundary of S are extracted, producing the table of `Ordered_endpoints` that is shown in Figure 3.16. The segment to which an endpoint belongs and its manhattan 3D distance to the origin of S, (located at (0,0)), are also recorded.

In the first iteration, `current_point` is $\mathtt{ep}_{2l}$. Segment s_2 is inserted in `Active_segments`. In the next iteration `current_point` is $\mathtt{ep}_{2r}$, a right endpoint. The `sweep_line()` (lines 13-14) looks for neighboring segments in `Active segments`. However, there are none (`Above_active_seg` and `Below_active_seg` are empty), therefore s_2 is the closest segment to S, and it is associated with the first interval $\mathtt{I}_{s_2}$ starting at (0,0). Next, `current_point` is $\mathtt{ep}_{2r}$, a special case because we have two consecutive endpoints in `Ordered_endpoints`



endpoint	ep 2l	ep 2r	ep 32l	ep 7l	ep 32r	ep 33l	ep 33r	ep 7r
segment	s_2	s_2	s_{32}	s_7	s_{32}	s_{33}	s_{33}	s_7
distance to origin	0	8	12	16	16	16	20	24
weight	0	0	4	0	4	4	4	0

Figure 3.16: `Ordered_endpoints` for the sweep

belonging to the same obstacle. In our implementation, before deleting s_2 we keep it in `Active_segments` for further testing.

When point $\texttt{ep}_{32l}$ becomes `current_point`, segment s_{32} is inserted after s_2 in the list of `Active_segments`. Although both segments are collinear, their placement in the list indicates that s_2 is traversed before s_{32} by the sweep-line. In this case, s_2 becomes `Below_active_seg` while `Above_active_seg` is empty.

Segments s_2 and s_{32} are collinear and $w_2 < w_{32}$. Their Voronoi edge computation, (lines 9-11 above pseudo-code), is described in section 3.3.3.3. Figure 3.17.a shows a graphical example, where the intersection of the Voronoi edge with S, (`VP`), is located at point (12,0). The event code determines that interval $\texttt{I}_{s_2}$ must be closed at `VP` and an interval $\texttt{I}_{s_{32}}$ must be opened for s_{32} starting at `VP` too. The dashed line segment with an open arrow in Figure 3.17 indicates that $\texttt{I}_{s_{32}}$ has not been closed. After this operation, s_2 is deleted from `Active_segments`.

When $\texttt{ep}_{7l}$ becomes `current_point`, s_7 is inserted in `Active_segments` after s_{32}, which implies that s_{32} is located below s_7. The Voronoi edge characterization for these two overlapping segments, where $w_7 < w_{32}$, is presented in section 3.3.3.2. In this particular case, the Voronoi "edge" intersects S at `VP` $= (16, 0)$ and bounds the dashed area in Figure 3.17.b. All the points inside it are equidistant to both s_7 and s_{32}. By default, our implementation choses the segment "below", (s_{32}), as the closest one to S. For this reason the current interval $\texttt{I}_{s_{32}}$ is not modified.

In the next iteration `current_point` is $\texttt{ep}_{32r}$, the right endpoint of s_{32}. `Above_active_seg`

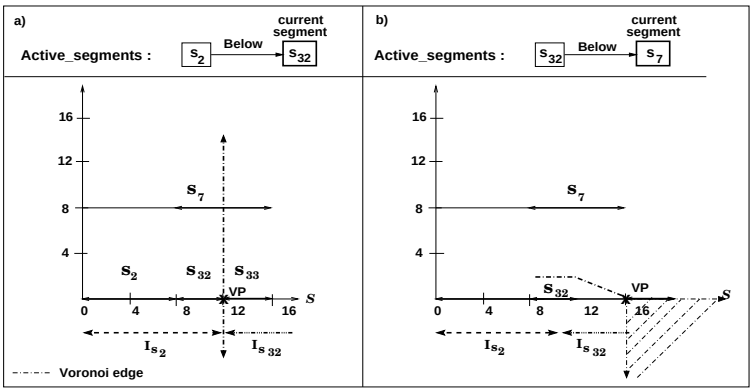


Figure 3.17: a)Voronoi edge between segments 2 and 32 b)Edge between segments 32 and 7

is s_7, but there is no `Below_active_seg` in `Active segments` (lines 12-15 above pseudo-code). Therefore, s_{32} is eliminated from the list.

Then `current_point` is ep_{33l}. s_{33} is inserted in `Active segments` "below" s_7. We have again two overlapping horizontal segments with $w_7 < w_{33}$ (see section 3.3.3.2), for this reason the Voronoi edge intersects S at `VP` = (8,0) (see Figure 3.18.a). In consequence, the event code indicates that s_{33} is closer to S than s_7. However, the relationship between segments s_{32} (owner of the current interval) and s_{33} remains unknown. An extra comparison between them is performed. The new `VP` is located at (12, 0) given that s_{32} and s_{33} are collinear and $w_{32} == w_{33}$ (see section 3.3.3.3). The Voronoi edge in Figure 3.18.b indicates that the interval (0,0)-(12,0) is closest to s_{32}, whereas the interval (12,0)-(16,0) is closest to s_{33}. Therefore, the event code determines that the current interval must be re-assigned from $\mathtt{I}_{s_{32}}$ to $\mathtt{I}_{s_{33}}$.

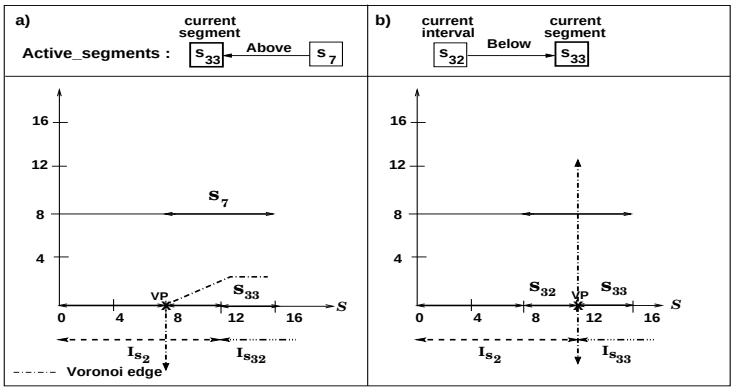


Figure 3.18: a) Edge for segments 33 and 7 b) Current interval is re-assigned from $\mathtt{I}_{s_{32}}$ to $\mathtt{I}_{s_{33}}$

For the next iteration `current_point` is $\mathtt{ep}_{33r}$, the right endpoint of $\mathtt{s}_{33}$. It is eliminated from `Active segments` because `Below_active_segment` does not exist. Now the list has only one element, $\mathtt{s}_7$.

Finally, `current_point` is the right endpoint $\mathtt{ep}_{7r}$. $\mathtt{s}_7$ has no neighbors in `Active_segments` and it is removed. The list empty because all the points have been processed. If at the end of the process there is an open interval (e.g. $\mathtt{I}_{33}$) it is closed. The intervals that were obtained for this example are shown in Figure 3.19.

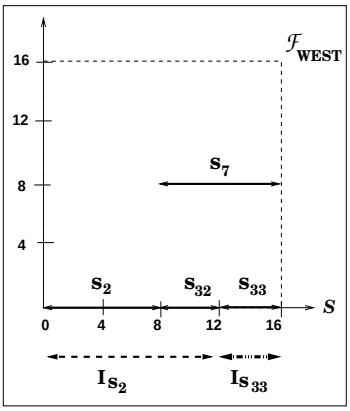


Figure 3.19: Intervals after the sweep

3.3.3 Computing the Voronoi Edge between two segments

In the previous section we presented the core of EODM building process: the sweep-line. In order to determine when to open or close an interval, the intersection of the Voronoi edge (between two obstacle segments) and the line separator S is obtained. The computation of the Voronoi edge is performed by algorithm `compute_voronoi()`, which is shown at the end of this subsection. Although we only describe the computation of the Vornoi edge for horizontal segments, the vertical ones are addressed using similar strategies.

Given two obstacle segments, let w_1 be the weight associated with the first segment ($\mathtt{s}_1$) and let w_2 be the weight associated with the second segment ($\mathtt{s}_2$). Recall that w_i is the Manhattan distance (obstacle $\mathcal{O}_i$ to $\mathcal{F}_i$) from the black node that gave rise to the Nodal Projection whose segment is s_i to the projection face. Our algorithm identifies three basic conditions: $\mathrm{w}_1 > \mathrm{w}_2$, $\mathrm{w}_1 < \mathrm{w}_2$, $\mathrm{w}_1 = \mathrm{w}_2$.

During the sweep, the Voronoi edge is computed only between two endpoints (e.g. that belong to two non overlapping segments) or using various "salient" points from the two competing segments. Both situations will be described with more detail in the next subsections.

3.3.3.1 Voronoi Edge for two endpoints of non-overlapping segments

The procedure `determine_Voronoi_for_endpoints()` performs the set of operations described in this subsection. Given two non overlapping segments, the shape of the edge is computed using the pair of endpoints (one belonging to each segment) that are closest to each other. Determining how the Voronoi edge divides the plane for this particular endpoints depends not only on their difference in x and y coordinates (dx and dy respectively), but also on their weights as described next.

- a) Case I : $w_1 > w_2$
  The horizontal subcases belonging to this group are shown in Figure 3.20. The vertical cases c_4, c_5 and c_6 are solved by symmetry.

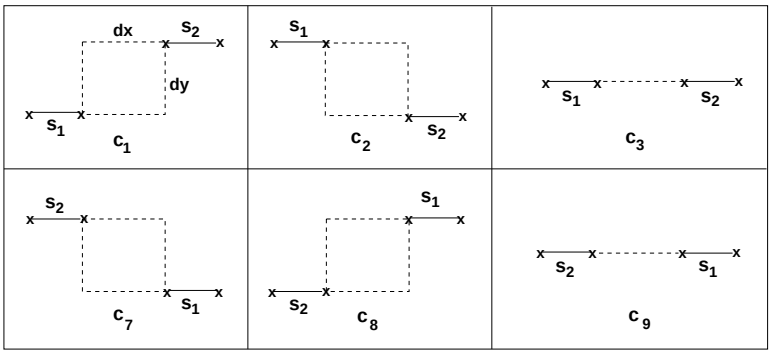


Figure 3.20: Case I: $w_1 > w_2$ without segment overlap

Let point `VP` be located at coordinates `(x,y)` in the projection face $\mathcal{F}_i$. The condition for `VP` to be part of the Voronoi edge between two segments s_1 and s_2 is that the distance from s_1 to it must be the same as the distance from `VP` to s_2. Figure 3.21.a shows the general form of the Voronoi edge for subcase c_1 without considering the weights of the segments. Notice that the shape of the edge is determined strictly by the coordinates difference components, dx and dy. Figure 3.21.b shows that the Voronoi edge is also influenced by the weights of the segments, however the effect is

only a shift in the vertical direction, (horizontal shift for vertical segments), and the shape of the edge remains the same. The equations of the Voronoi edge for the rest of the subcases in Figure 3.20 are computed in a similar fashion.

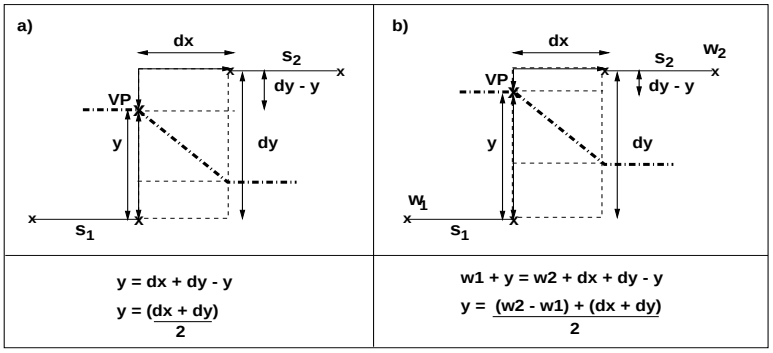


Figure 3.21: a)Voronoi edge, no weights b) As w_1, w_2 change, the edge shifts up or down

The different shapes of the Voronoi edge, for subcase c_1 are shown with a bold dashed line in Figure 3.22. The intervals that the edge generates for an horizontal separator are shown too. Notation I_{s_1}/I_{s_2} implies that the points in the interval are equidistant to both segments. If a sweep is performed in the north direction, the event code for Figure 3.22.b indicates that an interval for s_1 must be closed and a new one must be opened for s_2. The order is reversed is the sweep is performed in the south direction. Moreover, in Figure 3.22.d, the Voronoi edge will pass below the projection plane, therefore all the points in the plane are closest to s_1. If we use the equation in Figure 3.21.b to find the y-coordinate of VP, the value is larger than that of the vertices of s_2.

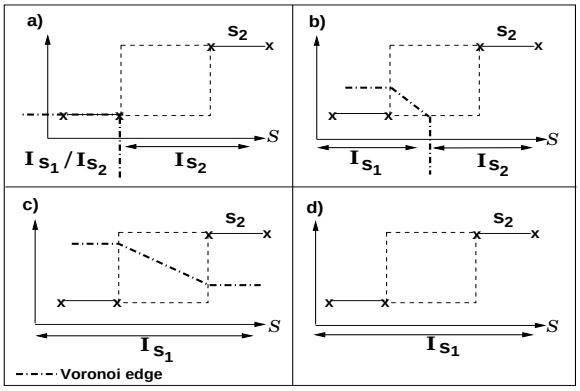


Figure 3.22: Shapes of the Voronoi Edge when computed for endpoints

When the competing segments are collinear, the Voronoi edge is simply a vertical line for horizontal segments, which can be shifted to the left or right according to the relationship between the weights. Figure 3.23 shows a graphic example and how the equation of the Voronoi edge is affected. For vertical segments, the Voronoi edge is horizontal and is shifted up or down according to the interaction between their weights.

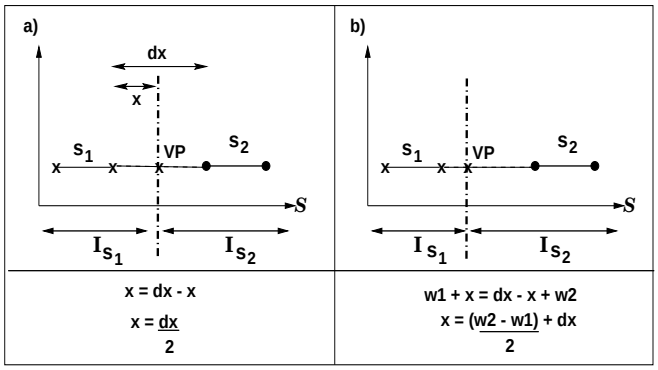


Figure 3.23: Voronoi edge for collinear segments: a) no weight b) weight

- b) Case II : $w_1 < w_2$

  Each subcase of Case I has an analogous one in Case II, the only difference is the switch in the weights of the segments. For this reason, these subcases are solved in a similar way as in the previous category, exploiting the geometry of each kind of circumstance.

- c) Case III : $w_1 = w_2$

  This case is the simplest because the weights do not influence the computation of the Voronoi edge. Case III is solved with the aid of `compute_V_edge_for_two_points()`, the Voronoi edge computation for the 2D EODM that was presented in Chapter 2, section 2.3.2.

  For horizontal collinear segments, the dx component is computed using the endpoints, one from each segment, that are closest between themselves. The Voronoi edge is the vertical line that passes exactly through the mid point of dx (see Figure 3.23.a). There is no shift due to the equality of the weights.

3.3.3.2 Voronoi Edge for overlapping non-collinear segments

When segments overlap there are two cases. If their weights are different, the algorithm `determine_Voronoi_for_segments()` is used to compute their Voronoi edge. In other case we use `determine_Voronoi_for_segments_equal_weight()`. In both cases, the edge computation is based on the endpoints operation `determine_Voronoi_for_endpoints()`, which was described in the previous subsection. Here, the three conditions for the weights are also present.

- a) Case I : $w_1 > w_2$

  The subcases contained in this group are shown in Figure 3.24. The Voronoi segment can have different configurations that directly depend on the weight of the segments and the extension of the overlap, see for example Figure 3.25, where the Voronoi edge of subcase c_0 (from Figure 3.24) is shown in a dashed line and it is computed for the horizontal separator S. The figure also shows the intervals along S and their associated closest segment. If an horizontal sweep is performed in the north direction, the event code for the segments in Figure 3.24.c is 12 (close the current interval for s_1 and open a new one for s_2). If the sweep is performed in the south direction, the event code is 21, causing the closure of the current interval for s_2 and the opening of a new interval for s_1. Notation I_{s_1}/I_{s_2} denotes that the 3D distance from s_1 to a particular interval along S is the same as the distance from that interval to segment s_2. In that event, if the current interval was associated with s_1 in a previous iteration of the `sweep_line()`, our algorithm discards s_2.

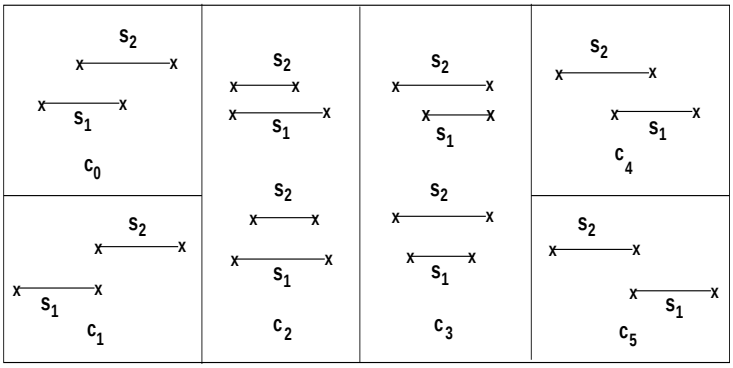


Figure 3.24: Case I: $w_1 > w_2$ for overlapping segments

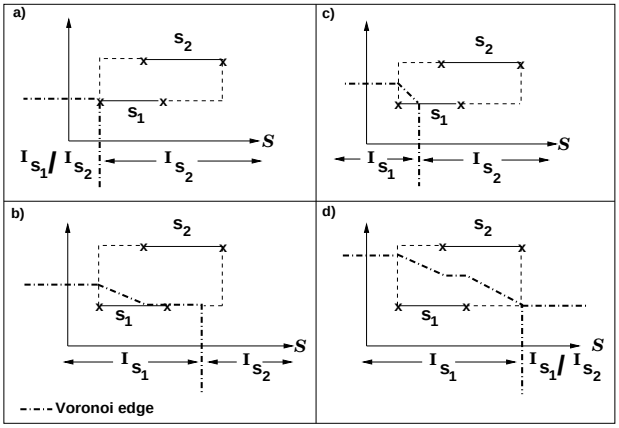


Figure 3.25: Possible Voronoi edges for overlapping segments for case c_0 in Figure 3.24

Determining the Voronoi edge for two overlapping segments $\mathtt{s}_1$ and $\mathtt{s}_2$ requires several calls to `determine_Voronoi_for_endpoints()`. We show the edge computation for Figure 3.25.d. The segments' endpoints are labeled as in Figure 3.26. The first point of the edge is found by `determine_Voronoi_for_endpoints(` $\mathtt{ep}_{11}$`,` $\mathtt{ep}_{21}$ `)` in Figure 3.26.a. Then, $\mathtt{ep}_{21}$ is projected over $\mathtt{s}_1$ to generate point $\mathtt{ep}_{21\pi}$. The procedure `determine_Voronoi_for_endpoints(` $\mathtt{ep}_{21}$`,` $\mathtt{ep}_{21\pi}$ `)` produces the second point of the edge in Figure 3.26.b. A similar process is performed for $\mathtt{ep}_{12}$, which is projected over $\mathtt{s}_2$, (see Figure 3.26.3). Finally, the last point of the edge, shown in Figure 3.26.d, is created by `determine_Voronoi_for_endpoints(` $\mathtt{ep}_{12}$`,` $\mathtt{ep}_{22}$ `)`.

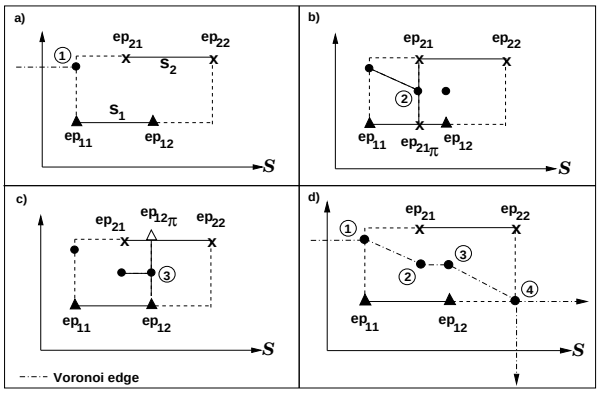


Figure 3.26: Computing Voronoi Edge for Figure 3.25.d

The reader can appreciate that exists certain resemblance between the Voronoi edges shown in Figure 3.22 and those from Figure 3.25. The x-coordinate of a Voronoi point `VP` on the projection face that forms part of the edge between two non collinear horizontal overlapping segments always has the general form:

$$\frac{d+w}{2} = x \tag{3.7}$$

where d is the difference in coordinates relationship (dx,dy) that defines the position of the edge and w is the weight relationship that represents the up/down shift. The y-coordinate of a `VP` is computed with a similar equation.

- b) Case II : $w_1 < w_2$
  The pair of segments falling in this category are solved using a similar strategy to that of Case I.

- c) Case III : $w_1 = w_2$
  Procedure `determine_Voronoi_for_segments_equal_weight()`, computes the Voronoi edge for non collinear segments. The strategy is similar to that of Case I. Essentially, at every segment endpoint, it applies the 2D EODM Voronoi edge computation, `compute_V_edge_for_two_points()`, previously presented in Chapter 2, section 2.3.2.

3.3.3.3 Voronoi Edge for overlapping collinear segments

The three main categories are presented next.

- a) Case I : $w_1 > w_2$
  For horizontal collinear segments, the Voronoi edge is always vertical. Figure 3.27 shows some examples of the edge, (in dashed lines), and the intervals that it generates. Figure 3.27.c lacks of a visible edge because, when the corresponding equation is applied, the x-coordinate of the Voronoi edge falls before the left endpoint of segment s_1. This indicates that segment s_2 is closest to the whole horizontal separator S.

  Let `x` represent the x-coordinate of the Voronoi point `VP`. Let `w` represent the lateral shift of the edge produced by the interrelation between the weights of the segments.

When two horizontal segments are collinear (see Figure 3.28), dy = 0 and equation 3.7 must be modified as:

$$\frac{dx + w}{2} = x \tag{3.8}$$

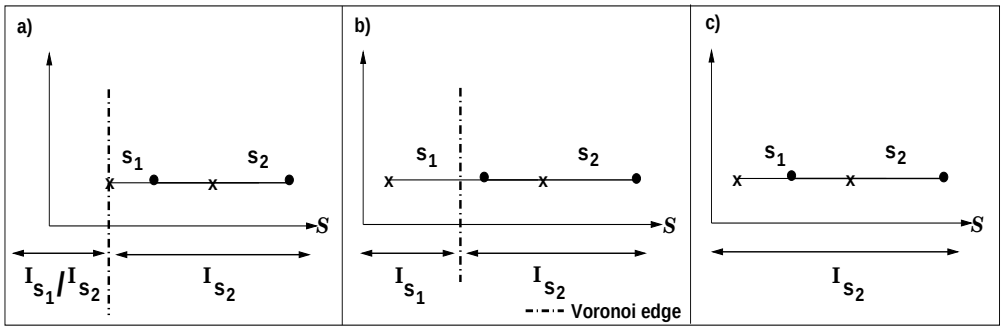


Figure 3.27: Possible Voronoi edge cases for collinear overlapping segments

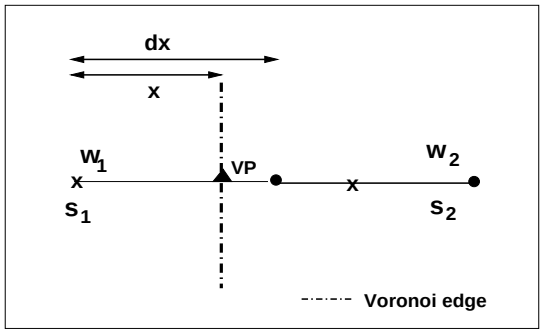


Figure 3.28: Computing the Voronoi edge for two collinear overlapping segments

- b) Case II : $w_1 < w_2$

  All the subcases that belong to this category are solved taking advantage of the geometry of each kind of situation using the methods described in Case I.

- c) Case III : $w_1 = w_2$

  For collinear segments, the Voronoi "edge" presents always the form shown in Figure 3.29. The segments' overlap generates a section for which S is closest to both of them. In the graphic example, the sweep line algorithm applied on a north separator, associates with s_1 the portion of S were the segments overlap.

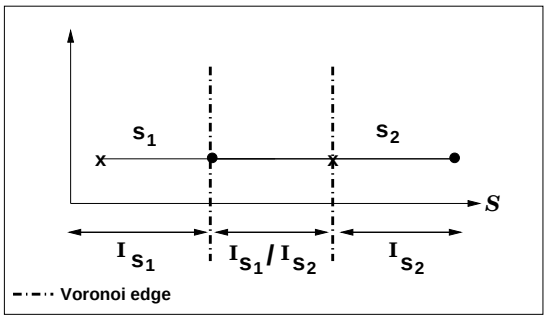


Figure 3.29: Voronoi edge for collinear overlapping segments with equal weight

Algorithm `compute_voronoi( s1, s2 )` is presented below. The procedures of lines 3, 4 and 5 return the intersection point of the Voronoi edge with the separator S, (`VP`), and the resulting event code (`e`) in order to update accordingly the intervals. The endpoints to test are chosen depending on the direction in which the sweep is performed.

```
compute_voronoi( s1, s2 )
Input :  segment s1, s2
Output:  point VP, /* intersection between Voronoi edge and S */
event_code e

1.  if ( overlap( s1, s2 ) )
2.       if ( s1->weight == s2->weight )
3.            return determine_Voronoi_for_segments_equal_weight( s1, s2, VP, e )
4.       else return determine_Voronoi_for_segments( s1, s2, VP, e )
5.  else return determine_Voronoi_for_endpoints( s1, s2, VP, e )
```

3.4 Algorithm `Make_EODM`

This section presents the complete algorithm for constructing an EODM. It requires an input file that specifies the dimension of the environment, as well as the locational codes of the obstacles. From the file information `create_simple_octree()` will return either a simple quadtree or octree and then the corresponding EODM construction algorithm will be called.

```
Make_EODM( FILE input_file )
Octree otree
Quadtree qtree

1.  Read from input_file the value DIMEN
2.  if (DIMEN == 3)
3.          otree = create_simple_octree( input_file, OCTREE )
4.          return Make_3D_EODM( otree, input_file )
5.  else
6.          qtree = create_simple_octree( input_file, QUADTREE )
7.          return Make_2D_EODM( qtree, input_file )
```

Chapter 4

Experiments

We implemented routines to build an EODM from an octree representation of the environment and to detect collisions of an L_1 spherical robot. Note that the former needs to be done once. Our results compare the DM, Octree, ODM and EODM in terms of memory requirements and speed of collision detection to quantify the various tradeoffs in 2D and 3D. Our routines are implemented in C and all the results we present are reported for a Sun Ultra 10 platform with 440 MHz UltraSPARCIIi processor.

4.1 Experiments for 2D models

We generated five environments to test the performance of EODM. Figure 4.1 shows a manhattan robot sphere in environment 3, which was also used by Jung [26].

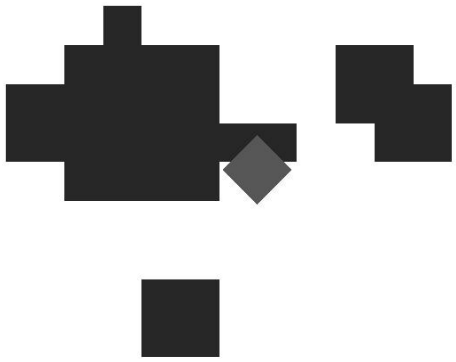


Figure 4.1: Robot manhattan sphere in a 2D environment

The memory needed (in Kb) for each approach is shown in Table 4.1 for each different environment (env. five is a chessboard pattern). EODM is seven to twelve times more memory intensive than ODM. However, EODM still provides a drastic improvement in memory needed (10%) compared to a DM. This percentage would be further reduced for finer discretizations and higher dimensions. We mention in passing that it took between 0.5 and 0.79 milliseconds to create the EODM from an octree for the five (256x256) environments, (including the chessboard), roughly two to seven times slower than an ODM.

Table 4.1 also shows the CPU time that it took to create an ODM and an EODM. EODM can be up to 36 times more expensive to compute than an ODM.

	Memory usage (Kb) 256x256				Creation time (m sec)	
Env	Octree	ODM	EODM	DM	ODM	EODM
1	0.064	0.080	0.620	65	0.1	0.2
2	0.548	0.570	6.860	65	0.9	8.0
3	0.744	0.780	8.780	65	1.2	23.5
4	0.904	0.962	9.260	65	1.1	39.6
5	0.996	1.060	10.220	65	0.9	27.8

Table 4.1: Memory usage/Creation Time for Quadtree, ODM, EODM and DM for five workspaces

The search algorithm used to test for collision using Octree and ODM is depth-first, starting at the root of the tree. The successors of this node will be evaluated in turn and added to a list of target (grey or black) nodes to be explored in the next iteration. The search is performed as deep in the tree as possible by visiting a node and then recursively performing depth-first on all the target nodes adjacent to that node. In the worse case, the Octree explores the entire tree. In comparison, in ODM, the search is not that exhaustive. It is only performed for a specific number of levels within the tree (minimum and maximum NSI). In average, only a small portion of the tree is explored.

For collision detection we used an L_1 spherical robot with a radius varying between one and two units (in pixels), and randomly changed its location 100 times. For each location and radius, we ran the test 100 times to discount fluctuations due to other processes. In order to compute the run times we used the function `ftime()`. The run times reported are the averages of these 10,000 runs for each environment. The cpu time includes the time needed to locate the white node in which the robot center lies. The collision detection times (in microseconds) and efficiency percentage for each approach are shown in Table 4.2. These

results show that EODM speeds up collision detection by a factor of two to seven against a conventional octree and by a factor of approximately two to five when compared with an ODM.

The maximum time observed for a single EODM test is about 0.097 milliseconds, when all the edges of the white node are tested for collision. Figure 4.2 shows the quantitative trade-offs of memory/CPU time for environment five.

256x256 Env	CPU time (μ sec) Octree	ODM	EODM	DM	Run-times ratio Octree over EODM	ODM over EODM
1	111.6	98.5	57.7	0.216	1.93	1.71
2	89.3	87.8	65.1	0.209	1.37	1.35
3	109.5	81.2	63.4	0.209	1.73	1.28
4	117.5	61.5	48.9	0.216	2.40	1.26
5	267.8	191.4	36.1	0.212	7.42	5.30

Table 4.2: CPU run-time for Quadtree, ODM, EODM and DM for the workspaces

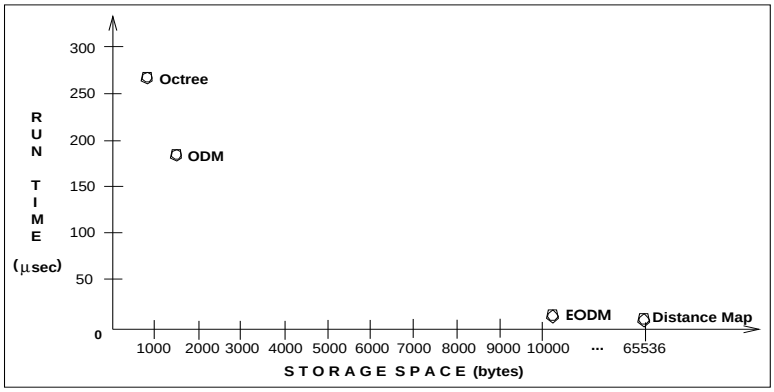


Figure 4.2: Qualitative Memory/Runtime trade-offs for env. 5

4.2 Experiments for 3D models

We used five 256 x 256 x 256 different environments. The memory needed for each environment, in MB, is presented in Table 4.3. EODM is about 20 times more memory intensive than an octree and an ODM. However, EODM only occupies 6% of the memory used by a DM.

Env	Memory usage (MB) 256x256 Octree	ODM	EODM	DM	Creation time (msec) ODM	EODM
1	0.116	0.140	2.236	16	1.0	26
2	0.240	0.330	5.704	16	1.2	112
3	0.368	0.458	8.792	16	1.7	363
4	0.408	0.606	9.112	16	3.7	358
5	0.444	0.624	10.496	16	5.8	500

Table 4.3: Memory usage/Creation Time for Octree, ODM, EODM and DM for five 3D workspaces

Table 4.3 also shows the CPU time that it took to create an ODM and an EODM for each environment. Starting from an octree, EODM can be way more expensive to compute than an ODM.

For collision detection we used an L_1 spherical robot with a radius varying between one and two units. We randomly changed its position 100 times. For each location and radius we ran the test 100 times to discard fluctuations due to other processes. We report the averages of these 10,000 runs for each environment. The CPU times include the time that it takes to locate the octree node in which the center of the robot falls. Collision detection times (in microseconds) and efficiency ratios for each approach are shown in Table 4.4. Our results show that EODM improves collision detection by a factor of 18 to 49 compared to a conventional octree and by a factor of 19 to 40 when compared to an ODM. The maximum time observed for a single sphere test takes about 94 microseconds, that is, when all the faces of the white node and all the edges of their corresponding cells are tested for collision. The run times we report were obtained using the function `ftime()`.

256x256x256 Env	CPU time (μ sec) Octree	ODM	EODM	DM	Run-times ratio Octree over EODM	ODM over EODM
1	1098.5	904.2	22.6	0.265	48.61	40.01
2	841.7	704.7	38.0	0.268	22.15	18.54
3	483.2	401.4	26.2	0.254	18.44	15.32
4	743.1	676.8	39.0	0.267	19.05	17.35
5	964.2	635.7	21.1	0.262	45.70	30.13

Table 4.4: CPU run-time for Octree, ODM, EODM and DM for the 3D workspaces

Figure 4.3 shows graphically the run time for a subset of collision tests performed in

environment 5 for all the approaches. The horizontal axis simply represents different locations for the robot sphere. The circles represent the CPU time of DM, the octree time is represented by rhomboids, the ODM time as squares, and finally, the EODM time is represented by triangles. Notice the almost "constant" behavior or our EODM. Figure 4.4 shows the quantitative trade-offs of memory/CPU time for environment five (a chessboard pattern).

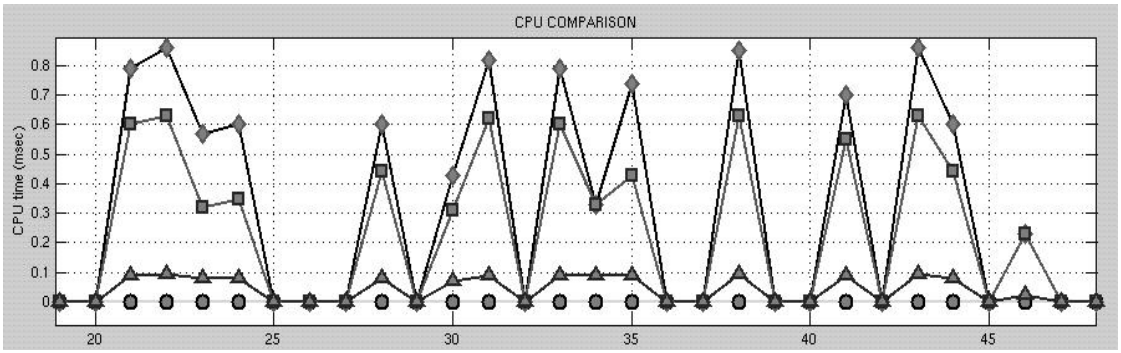


Figure 4.3: Memory/Runtime trade-offs for 3D env. 5

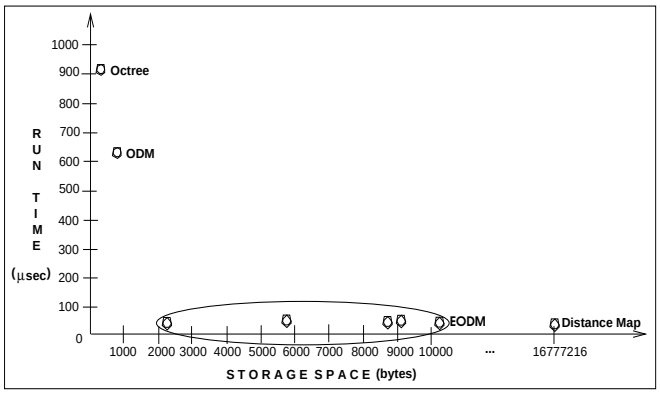


Figure 4.4: Qualitative Memory/Runtime trade-offs for 3D env. 5

Finally, we compared EODM, ODM and Octree's cpu performance using the seven degree of freedom robot model from Jung [26] covered by 159 spheres. Figure 4.5 and Figure 4.6 show the L_1 arm, whereas Figures 4.7 and 4.8 show the robot in a simple environment. We tested four robot configurations, two of them in collision and two free of collision. Although for a strict COLLISION/FREE answer the collision algorithm should stop as soon as a sphere is detected in collision, we report on all the spheres in collision just for

testing purposes. The results are shown in Table 4.5. In the first row, EODM is faster than the other approaches because the configuration of the robot falls mostly inside white nodes. EODM provides a significant improvement in performance (66-94%, average 75.5%) compared to the octree and (51-89%, average 67.5%) compared to an ODM. DM is not reported because its cpu time would be 159 times the single sphere cpu time.

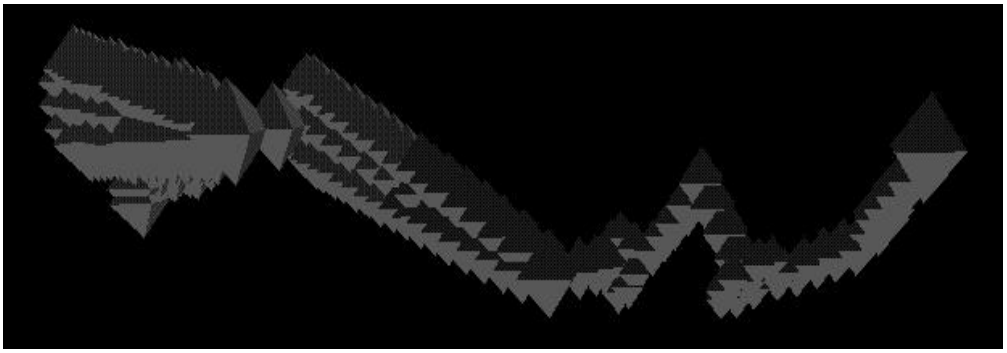


Figure 4.5: Robot covered by 159 L_1 spheres

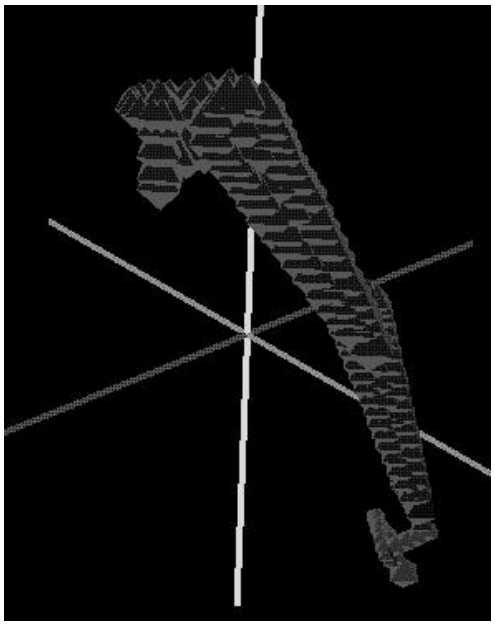


Figure 4.6: Another view of the robot

CPU time (msec) 256x256x256									
Robot Conf.	Spheres in collision	Total CPU time required			Average time per sphere			Improvement (Ratio)	
		Octree	ODM	EODM	Octree	ODM	EODM	Octree vs EODM	ODM vs EODM
1	63	243	231	15	1.5283	1.4528	0.0943	16.20	15.40
2	147	50	35	17	0.3145	220.1	0.1069	2.94	2.06
3	none	80	74	20	0.5031	465.4	0.1258	4.00	3.70
4	none	77	56	26	0.4843	352.2	0.1635	2.96	2.15

Table 4.5: CPU run-time with 4 robot configurations (159 spheres)

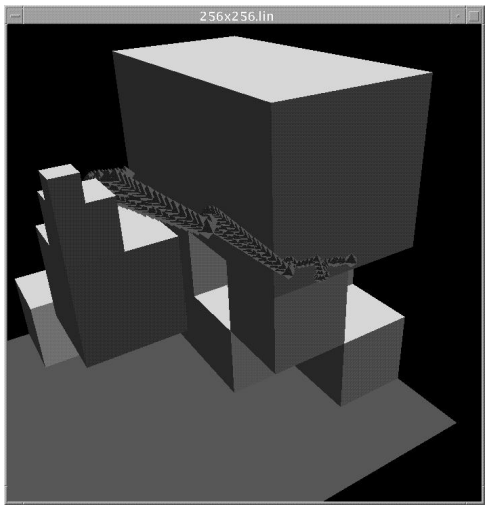


Figure 4.7: Robot arm free of collision

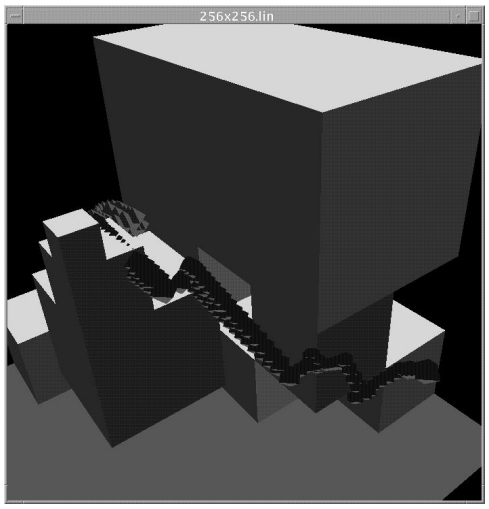


Figure 4.8: Robot arm in collision

Chapter 5

Conclusions and Future Work

We introduced the Extended Octree Distance Map (EODM) for efficient collision detection in static environments. EODM is a systematic hierarchical representation for distance maps. It utilizes an octree as the base representation. Along the perimeter of each white node in the octree, it stores a distance function that represents the distance of each boundary point of the white node to the obstacle closest to that point. With an EODM, collision detection becomes a simpler and faster process given that only a comparison between the radius of the robot and the distance values, (indexed by the robot projection), over the boundaries of the node in which the robot lies is needed.

We presented algorithms for creating the EODM as well as algorithms to use it for collision detection. EODM is constructed once as an off-line process and then repeatedly used for collision detection queries.

Collision detection with an EODM is a constant time operation once the node in which the center of the robot is contained has been found. In terms of storage, EODM requires more memory compared to a standard octree and an ODM introduced in [26]. The storage space occupied by an EODM on average is proportional to the surface area of the objects in the scene (as in an octree, but the constant of proportionality would be larger) and not to the volume.

Our experiments in 2D and in 3D have shown that EODM (i) decreases collision detection time dramatically over other ocree-based collision approaches and (ii) provides a reduction in memory storage compared to a voxel Distance Map. This characteristics are shown qualitatively in Figure 5.1.

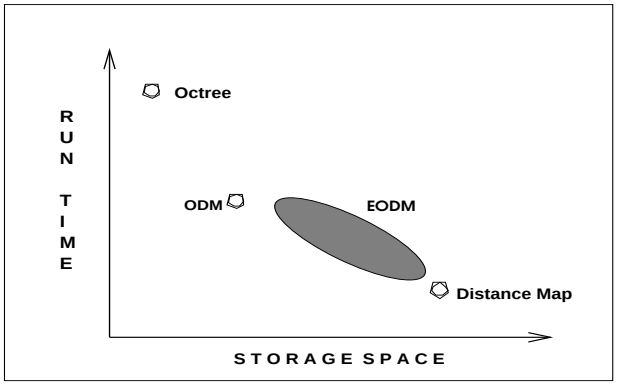


Figure 5.1: Memory-Runtime trade-offs

5.1 Future Work

We would like to explore the advantages/disadvantages of our work compared to other hierarchical representations, such as k-d trees, the BAR model from Duncan et. al [9] which combines octrees and k-d trees, and other hierarchical structures. Octrees are good when the data, in this case the obstacles, are uniformly or randomly distributed [38]. However it is well known that they can generate a lot of nodes that contain no obstacles, see for example Figure 5.2, which produces an expensive deep tree for collision detection. K-d trees [9, 38] and other Binary Space Partition [6] based models produce fewer white nodes. Their advantage is not only that they reduce the storage space, but also that they require a shallower tree that can improve the search during collision detection.

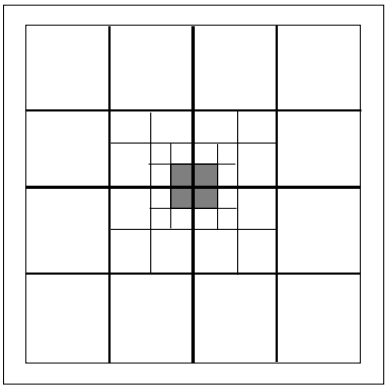


Figure 5.2: Unnecessary white cells

In the same line of work we would like to experiment with the previous models in order to provide a "fair" comparison with OBBTrees [31] and Sphere-based [8] models. Gipson, in his empirical experiments with the Motion Planning Kit (MPK) [13], shows that VCollide takes approximately 1 millisecond solving collision queries for the same robot model used in this work.

EODM can be modelled using L_2 distance too. However, the shortest path between the robot and the closest obstacle point does not necessarily pass through the projection of the robot's center onto the faces of the node that contains it. In order to obtain the closest obstacle we need to compute the distance from the robot's center to *all* the obstacle points that are closest points along the boundary of the node that contains the robot and select the one with the minimum value. Certainly L_2 metric will reduce the number of spheres of our robot model, however, collision detection would be a slower process, not a constant time operation.

With respect to our current implementation, we mentioned that EODM is built using a global approach, where all the visible obstacles from the faces of each white node W are projected at once. There's also the option of using a local approach where only the obstacles within the parent cell W are projected over its boundary faces. The local approach may be faster to build and update each time a robot scans a new portion of the space in Sensor-based motion planning. However it is slower than the global approach for collision detection because the distance information has to be retrieved and tested against the robot at each level of the tree until the node that contains the robot's center has been found.

EODM stores redundant information. We would like to exploit the locality of the model, (coherence) and remove the unnecessary line separators. For example, Figure 5.3.a shows a white cell c_i. Chances are that the intervals computed for the east direction of the vertical separator S_1 will not change in the area comprised between S_1 and S_3 making S_2 unnessential (Figure 5.3.b).

Currently, we store intervals along the line separators, which represent the set of boundary points of white cells that are located closest to a particular obstacle. We still need to compute the distance from a point in the interval to its closest obstacle during collision detection time. In 3D, this process consumes most of the time. We propose instead to store the perimeter function $d_p(s_1, s_2)$ itself and the two bounds of the white cell boundary over which the function applies in the interval. Any point falling in between the bounds can be evaluated right away, thus reducing the distance computation time. The question now

becomes how to model and store the perimeter function more efficiently.

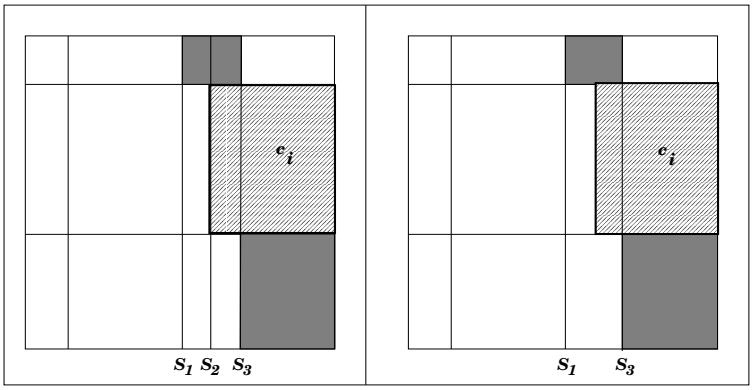


Figure 5.3: a)Storage of redundant information b)Line separators removal

Appendix A

Procedure Connect_3D_EODM()

Let sep_tree be the generic name for either an horizontal or a vertical separator tree. Connect_3D_EODM() preserves the connectivity between the white nodes of the EODM and the projection faces. The pseudo-code is presented next.

```
Connect_3D_EODM(otree, parnode, lvl, MAX_CHILDREN)
Input :  Octree otree, octree_node parnode, node_level lvl
Output:  EODM

1.  lvl = lvl+ 1
2.  if ( COLOR(parnode) == BLACK )
3.          return;
4.  for (chnum = 0; chnum < MAX_CHILDREN; chnum++)
5.          chptr = parnode->child[chnum]
6.          if ( COLOR( chptr ) == WHITE ) /* WHITE child node, find separators */
7.              for (face_dir = NORTH to face_dir = BACK )
8.                  if ( face_dir is NORTH || SOUTH )
9.                       sep_tree = XY_separator
10.                 else if ( face_dir is EAST || WEST )
11.                           sep_tree = XZ_separator
12.                      else sep_tree = YZ_separator
13.                 if ( face_dir is SOUTH || WEST || FRONT )
14.                      node_dir = RIGHT
15.                 else node_dir = LEFT
16.                 insert_separator( sep_tree, chptr )
17.                 link_white_node( chptr, sep_tree, node_dir, face_dir )
            /* GRAY or BLACK child node */
18.         else Connect_3D_EODM( otree, chptr, lvl, MAX_CHILDREN )
```

Appendix B

Linking the white cubes with the separators

Each white node of the EODM is associated with its adjacent face separators in each direction (FRONT, BACK, NORTH, WEST, SOUTH, EAST). Procedure `link_white_node()` is presented next.

```
link_white_node( W, sep_tree, Sep_dir, Face_dir)
Input :  octree_node W, Separator sep_tree, direction Sep_dir, direction Face_dir
          Separator Snode
Output :  Boundaries of W associated with a projection plane
1.  value = get_reference_point( W, Face_dir )
2.  Snode = get_separator( sep_tree, value )
3.  W->boundary[ Face_dir ] = Snode
4.  for (i=0; i < NUM_FACES; i++)
5.         if ( W->boundary[ i ] == NULL )
6.                    return
```

For example, `link_white_node(` W, XZ_separator, RIGHT, WEST) implies that a vertical separator in the XZ plane is linked to the WEST boundary of a white node and the $\hat{n}$-visible 3D obstacles to project are located to its RIGHT side.

Procedure `get_separator()`, returns the id of the corresponding adjacent face separator to each boundary of the white node. `NUM_FACES`, the number of faces of a white node, is only used for clarity.

Appendix C

User's Guide

We include a guide to use our system in order to generate an EODM in two and three dimensions. Moreover, we describe how to use it to solve collision detection queries.

Input

The system reads the workspace information from a file with the extension **.lin*. The format of this file is as follows:

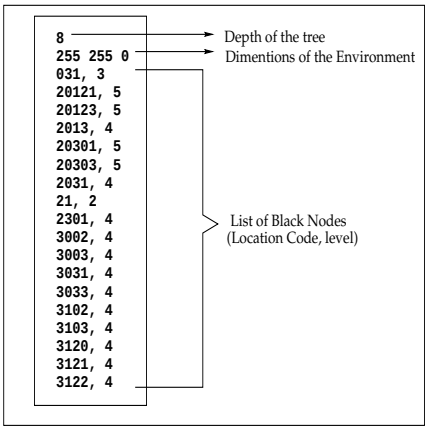


Figure C.1: Format of the input file

Where the first line indicates the maximum number of levels in the tree. The second line indicates size of the environment in the x, y and zth dimensions respectively. In this case a 2D EODM of 256 $\times$ 256 units will be created because the value of the zth coordinate is zero. The following lines in the file are a list of the black nodes that correspond to obstacles

in the environment. We assume that this information has already been obtained by the sensors. The list is composed of pairs (`LC`, `lvl`), where `LC` is the Locational Code of the black node and `lvl` is the level of that node in the tree. The list in Figure C.1 produces the environment shown in Figure C.2.

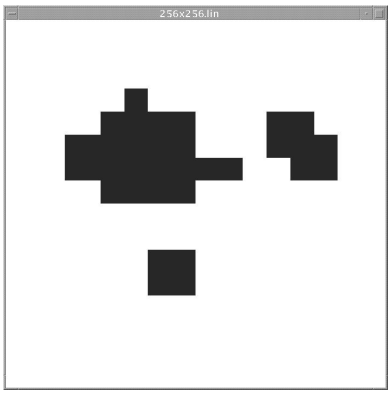


Figure C.2: 2D environment from file 256x256.lin

2D EODM

To invoke the program, type in a console window the command:

> eodm2d < *filename.lin* >

where *filename.lin* is the input file that contains the description of the environment of the robot.

The system allows you to test a robot using a graphical interface or using a command menu. If you choose to use the interface, to place a robot in the environment press the left button of the mouse. That will define the location of the center. Then, move slightly around and press the middle button, which defines the length of the radius of the robot. Now, by pressing the right button a menu appears that allows to test the robot for collision or exit the system (see Figure C.3). The result of the collision test, the closest obstacle and the minimum distance between the robot and the closest obstacle are displayed in the console.

If the graphical interface is not used, the statistics (memory used) of the environment will be shown, and then you will see the following menu:

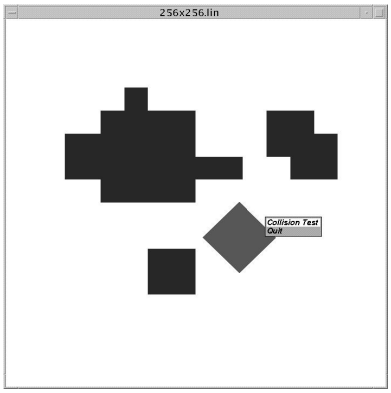


Figure C.3: Testing a 2D robot for collision

```
        COLLISION DETECTION MAIN MENU
            ------------------
Choose:
[1] Enter robot positions/sizes from keyboard
[2] Get robot positions/sizes from file
[3] Generate/Test multiple random robot positions
[4] Perform collision detection
[7] Quit
          --> __
```

Option one allows to input the robot information (center and radius) using the keyboard. Option two reads that information from a file specified by the user. Option three allows to generate multiple random robot locations in the 2D environment and test each one of them for collision. This information can be stored to a file and used for future collision test accessing option 3 too. The cpu statistics of each set of collision tests can be stored to a file if desired. Option four allows to test for collision the current robot information (obtained from options one or two).

3D EODM

The 3D EODM can be executed by typing in a console:

$$> \text{eodm3d} < \textit{filename.lin} >$$

where *filename.lin* is the file that describes the environment of the robot. A simple 3D

environment is shown in Figure C.4. Both black and white nodes are visible, as well as a single robot sphere on the upper right corner of the octree. The robot information is read from the file *robot.dat* by default. To test the robot sphere for collision, press the right button of the mouse and choose the option "Collision Test: Single Sphere". The status, the closest obstacle and the minimum distance will be displayed on the console.

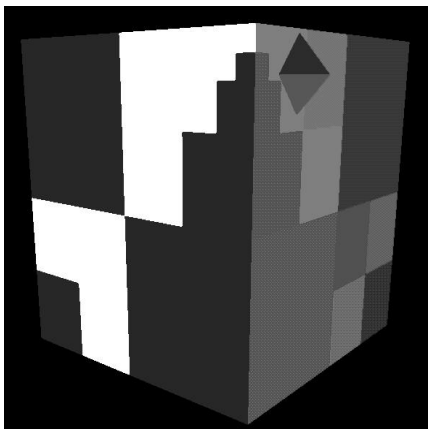


Figure C.4: Testing a 3D robot for collision

If a robot arm is desired, press the right button of the mouse and choose the option "Draw Robot Arm" (see Figure C.5). Because it is difficult to visualize, the white nodes of the EODM will disappear.

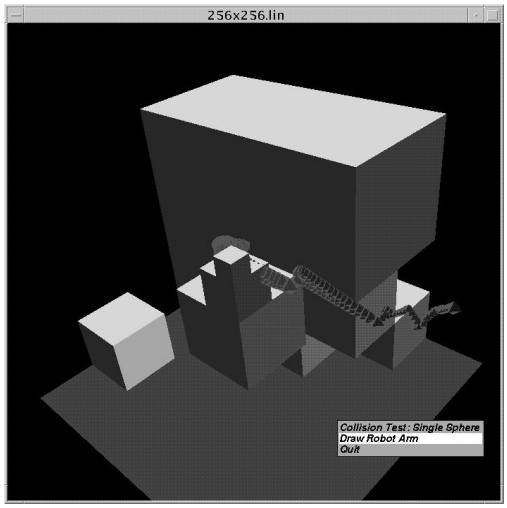


Figure C.5: Testing a robot arm for collision

At that moment, the system prompts the user to input the file that contains the robot arm information. For this example, we used the file *robot3-6.dat* generated by Jung [26]). Once the robot is drawn, each sphere is tested for collision automatically. The cpu statistics and collision results can be stored to a file chosen by the user.

In case the graphical interface is not used, the functionalities of the console operated menu (shown below) are the same as the ones for the 2D EODM, with the exception of option five. It allows to read and test for collision the set of spheres of a robot arm. The cpu statistics and status can be saved to the file *eodm.mat*. This file can be visualized in matlab in order to produce the plots shown in Chapter 4.

```
      COLLISION DETECTION MAIN MENU

    ---------------------------------
[1] Enter robot positions/sizes from keyboard
[2] Get robot positions/sizes from a file
[3] Generate/Test multiple random robot positions
[4] Perform collision detection
[5] Test robot arm spheres file
[6] Quit
Select an option --> __
```

Bibliography

[1] Arimoto, S., H. Noborio, S. Fukuda, and A. Noda. A feasible approach to automatic planning of collision-free robot motions. In *International Symposium on Robotics Research*, pages 479–488. B.R.R. Bolles editors. MIT Press, 1988.

[2] Barraquand, J. and J.C. Latombe. Robot motion planning: A distributed representation approach. *International Journal of Robotics Research*, 10(6):628–649, 1991.

[3] Brabec, Frantisek and Hanan Samet. Spatial index demos. Collection of java applets based on the algorithms of Hanan Samet. [http://www.cs.umd.edu/ brabec/quadtree]. (January, 2001).

[4] Chen, P.C. and Y.K. Hwang. SANDROS: A motion planner with performance proportional to task difficulty. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, pages 2346–2353. Nice, France, 1992.

[5] Cohen, Jonathan, M.C. Lin, D. Manocha, and K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *1995 Symposium on Interactive 3D Graphics*, pages 189–195, 1995.

[6] de Berg, M. Linear size binary space partitions for uncluttered scenes. *Algorithmica.*, 28:353–366, Springer-Verlag, New, York Inc. 2000.

[7] del Pobil, A.P. Sd motion planning based on a spherical hierarchical representation. In F. Cantú and H. Terashima, editors, *Proc. of the International Symposium on Artificial Intelligence*, pages 91–97. Ed. Limusa, México, 1991.

[8] del Pobil, A.P. and Miguel A. Serna. A new representation for collision avoidance and detection. In *IEEE International Conference on Robotics and Automation*, pages 246–251. Nice, France, May, 1992.

[9] Duncan, Christian A., Michael T. Goodrich, and Stephen Koburov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. In *10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 300–309, 1999.

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

[11] Elfes, Alberto. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, June 1990.

[12] Foley J.D., A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice. Second Edition in C.* Addison Wesley, 1996.

[13] Gipson, Ian, Kamal Gupta and M.A. Greenspan. MPK: An Open Extensible Motion Planning Kernel. InJournal of Intelligent Robotic Systems, 8(18):433–443, 2001.

[14] Greenspan, Michael and Nestor Burtnyck. Obstacle count independent real-time collision avoidance. In *IEEE International Conference on Robotics and Automation*, pages 1073–1078, 1996.

[15] Gupta Kamal K. *Motion Planning: Overview and State of the Art*, pages 3–8. Practical Motion Planning in Robotics: Current Approaches and Future Directions. John Wiley and Sons, West Sussex, England, 1998.

[16] Hayward, Vincent. Fast collision detection scheme by recursive decomposition of a manipulator workspace. In *Proc of the IEEE Intl Conference on Robotics and Automation*, pages 1044–1049, 1986.

[17] Hearn Donald and Pauline M. Baker. *Computer Graphics, C version. Second edition.* Prentice-Hall, USA, 1997.

[18] Hoff III, Kenneth, John Keyser, Ming Lin, Dinesh Manocha and Tim Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Computer Graphics. Annual Conference Series*, 33:277–286, 1999.

[19] Hoff III, Kenneth, T. Culver, J. Keyser, MC. Lin, and D. Manocha. Interactive motion planning using hardware-accelerated computation of generalized voronoi diagrams. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation*, pages 2931–3726. San Francisco, CA., April 2000.

[20] Hudson, T., MC. Lin, J. Cohen, S. Gottschalk, and D. Manocha. VCOLLIDE: Accelerated collision detection with VRML. In *Proceedings of VRML*, 1997.

[21] Hunter G.M. *Efficient Computation of Data Structures for Graphics.Ph. Thesis.* PhD thesis, Department of Electrical Engineering and Computer Science. Princeton University, Princeton, NJ, 1978.

[22] Hwang, Y.K. and Narendra Ahuja. Gross motion planning - a survey. *ACM Computing Surveys*, 24(3):219–291, September, 1992.

[23] Hwang Y., P.G. Xavier, P.C. Chen, and P.A. Watterberg. *Motion Planning with SANDROS and the configuration space toolkit*, pages 55–78. Practical Motion Planning in Robotics: Current Approaches and Future Directions. John Wiley and Sons, West Sussex, England, 1998.

[24] Jiménez, P., F. Thomas, and C. Torras. 3d collision detection: A survey (2000). [http://citeseer.nj.nec.com/431815.html]. (August, 2001).

[25] Jung, Derek and Kamal Gupta. Octree-based hierarchical distance maps for collision detection. *Journal of Robotic Systems, John Wiley*, 14(11):789–806, 1997. A version also appeared in IEEE International Conference on Robotics and Automation. Minneapolis. 1996.

[26] Jung Derek. Range image integration and hierarchical distance maps for sensor-based collision detection and path planning. Master's thesis, School of Enginering Science. Faculty of Applied Science. Simon Fraser University., Canada, August, 1997.

[27] Kavraki, L.E., P. Švestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high dimensional configuration spaces. *IEEE Trans. Robot. Autom.*, 12:566–580, 1996.

[28] Kavraki, L.E. and J.-C. Latombe. *Probabilistic roadmaps for robot path planning*, pages 33 – 53. Practical Motion Planning in Robotics: Current Approaches and Future Directions John Wiley and Sons. West Sussex, England, 1998.

[29] Laubach, S.L. and J.W. Burdick. Wedgebug: A sensor-based path planner for planetary microrovers. In *IEEE Intl Conference on Robotics and Automation*, Workshop W8:

Integrating Sensors with Mobility and Manipulation. Organizers: Kamal K. Gupta, Angel del Pobil, Howie Choset, April 2000.

[30] Lin, Ming C. and John F. Canny. Fast algorithm for incremental distance calculation. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 1008–1014, 1991.

[31] Lin, M.C., Gottschalk S. and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the ACM-SIGGRAPH*, pages 171–180, New Orleans, L.A. August 1996.

[32] Lin, M.C. and S. Gottschalk. Collision detection between geometric models: A survey. In *Proceedings of IMA Conference on Mathematics of Surfaces*, 1998.

[33] Meagher D. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3d objects by computer. Technical report ipl-tr-80-111. Image Processing Laboratory. Rensselaer Polytechnic Institute, Troy, NY, October 1980.

[34] Mirtich Brian. *Efficient Algorithms for Two-Phase Collision Detection*, pages 203–223. Practical Motion Planning in Robotics: Current Approaches and Future Directions. John Wiley and Sons, West Sussex, England, 1998.

[35] Moravec, H.P. and A. Elfes. High resolution maps from wide angle sonar. *IEEE International Conference on Robotics and Automation*, pages 138–145, 1985.

[36] Quinlan, Sean. Efficient distance computation between non-convex objects. *IEEE International Conference onRobotics and Automation*, 1994.

[37] Samet Hanan. *Applications of spatial Data Structures: Computer Graphics, Image Processing and GIS.* Addison-Wesley Publishing Company, Reading, MA, 1990.

[38] Samet Hanan. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley Publishing Company, Reading, MA, 1990.

[39] Samet, Hanan. The quadtree and related hierarchical data structures. *ACM Comp. Surveys*, 16(2):187–260, June 1984.

[40] Schneier, M. Path-length distances for quadtrees. *Information Sciences*, 23:49–67, 1981.

[41] Yu, Yong and Kamal Gupta. On sensor-based roadmap: a framework for motion planning for a manipulator arm in unknown environment. In *IEEE/RSJ International Conference on Intelligent Robot and System*, pages 1919–1924, 1998.