Robot Obstacle Avoidance using the Kinect

RASOUL MOJTAHEDZADEH

KTH Computer Science
and Communication

Master of Science Thesis
Stockholm, Sweden 2011

Robot Obstacle Avoidance using the Kinect

R A S O U L M O J T A H E D Z A D E H

Master's Thesis in Computer Science (30 ECTS credits)
at the Systems, Control and Robotics Master's Program
Royal Institute of Technology year 2011
Supervisor at CSC was John Folkesson
Examiner was Danica Kragic

TRITA-CSC-E 2011:107
ISRN-KTH/CSC/E--11/107--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

Abstract

Kinect for Xbox 360 is a low-cost controller-free device originally designed for gaming and entertainment experience by Microsoft Corporation. This device is equipped with one IR camera, one color camera and one IR projector to produce images with voxels (depth pixels). This additional dimension to the image, makes it a tempting device to use in robotics applications. This work presents a solution using the Kinect sensor to cope with one important aspect of autonomous mobile robotics, obstacle avoidance.

Modeling the environment based on the point cloud extracted from the depth image data as well as an obstacle avoidance method using the straight line segments and circle arcs, were the main focus of the thesis. The environment is represented by a set of polygons and the obstacle avoidance algorithm attempts to find a collision-free path from the current position of the robot to the target pose subject to the shortest possible path considering a safeguard.

The whole algorithm was implemented and simulated in Matlab successfully. However, it is partially implemented in C++ and integrated into the current software architecture of the project CogX running at CVAP/CAS department, KTH. A mobile robotic platform based on Pioneer P3-DX which was equipped with a LIDAR module which the previous obstacle avoidance algorithm relied on. The shortcoming was not being able to detect all types of obstacles. This motivated using the Kinect sensor to extend the ability to detect obstacles and hence to improve the local path planner and obstacle avoidance components in the software.

Acknowledgements

I would like to express my gratitude to *Dr. John Folkesson*, my main advisor, for his supervision, great engagement and guidance. I owe many thanks to *Dr. Patric Jensfelt* for his great effort in directing the program Systems, Control and Robotics during two years of my master studies. I would also like to thank *Alper Aydemir* for helping me in implementation part of the thesis.

Contents

Chapter 1

Introduction

Robotic systems with mobility should be able to explore the environment autonomously. Human being, animals and even insects are able to explore their surrounding environment safely. They can move around and carry out different tasks without colliding with static and dynamic obstacles. The main perception involved in this task is *vision* when mixed with other kinds of sensing such as *touch* and *hearing* makes a perfect input for the biological learning system to learn how to optimally perform obstacle avoidance in unstructured environments. They silently create a map of the environment, plan a safe path and execute a real-time obstacle avoidance. Adding such capability is an essential requirement for a fully autonomous mobile robot to be able to carry out its tasks.

As some examples, domestic robots specifically designed to clean a house cannot handle the assigned tasks without an effective obstacle avoidance and path planning system as well as mapping the environment. Some of the commercially available domestic robots are shown in Figure 1.1.

The importance of sensing motivated the robotics community to examine different kinds of sensors mostly ranging from sonars and laser range scanners to stereo vision systems. Kinect for Xbox 360 is a low-cost vision device equipped with one IR camera, one color camera and one IR projector to produce RGB images as well as voxel (depth-pixel) images. Albeit the main aim of original design by Microsoft Corporation was for video gaming and entertainment but a great interest in the robotics community to examine their capabilities soon started after the product had been released. The ability of the device to add one extra dimension to the images is the key feature which makes it potentially invaluable as a low-cost sensor in robotics systems with wide range of applications.

The main objectives of this work were to construct a model of the environment by using the data from the Kinect, given this model, designing a path planner to build a non-colliding safe path and finally implement (partially/completely) the algorithm in C++ on the mobile robot platform Pioneer P3-DX, dedicated to the project CogX running at CVAP/CAS department, KTH.

The algorithms developed for both the modeling and obstacle avoidance were

(a) First generation Roomba vacuum

(b) Home Assistant Robot AR

(c) daily life mobile robot assistant Care-O-bot III

Figure 1.1: Some domestic robots with different level of perception, actuation and intelligence designed to assist people in domestic environments.

implemented and simulated in Matlab software. The implementation on the robot included integration to the current software architecture which utilized LIDAR sensor data only to build a 2D grid map as well as a specifically metric map using by obstacle avoidance component. The lack of ability to detect obstacles standing in higher or lower positions than the LIDAR was a potential danger for the robot to collide with. Adding the extra information receiving from the Kinect sensor and fuse to the laser data was the main implementation idea performed.

1.1 Outline

In chapter 2 an introduction to the Kinect sensor with more details on hardware and software is given. Chapter 3 discusses modeling of the environment based on the depth data streaming from the Kinect sensor and presents some theory and results. Chapter 4 describes in detail the methodology developed for pre-computation of a non-colliding path connecting the start and goal poses. Implementation of the algorithm is discussed in chapter 5. Problems we faced during the project and the conclusion are summarized in chapter 6.

Chapter 2

Kinect - A 3D Vision System

Microsoft Corporation announced and demonstrated a new add-on device for the Xbox 360 video game platform named as *Project Natal* (which later called as *Kinect sensor*) in June 2009 which attracted the robotics community to evaluate it as a potentially valuable device. Kinect lunched on November 2010 in north America and Europe, though. The amount of official information published by the manufacturer, Microsoft is less than required for such a research on the device. Hence, the following content of information is mostly derived by sources who obtained the data through reverse engineering efforts or so-called hacking.

2.1 Hardware

Kinect is based on software technology developed internally by Rare [31], a subsidiary of Microsoft Game Studios owned by Microsoft, and on range camera technology by PrimeSense [33], which interprets 3D scene information from a continuously-projected infrared structured light. The depth sensor consists of an infrared laser projector combined with a monochrome CMOS sensor, which captures video data in 3D under any ambient light conditions [34]. However, our experience during dealing with the Kinect sensor revealed that it cannot return any depth data for very dark or bright areas. The device is also equipped with an RGB camera located between the IR projector and IR camera. This camera has no role in depth measurement.

The Kinect sensor outputs video at a frame rate of 30 Hz. The RGB video stream uses 8-bit VGA resolution (640 $\times$ 480 pixels) with a Bayer color filter, while the monochrome depth sensing video stream is in VGA resolution (640 $\times$ 480 pixels) with 11-bit depth, which provides 2,048 levels of sensitivity. The sensor has an angular field of view of 57° horizontally and 43° vertically. The Kinect sensor has a practical ranging limit of 1.2 - 3.5 m distance when used with the Xbox software [34]. Figure 2.1 shows the device and some annotation indicating the cameras and projector.

Figure 2.1: The Kinect sensor.

2.1.1 Depth measurement

Although the developer of depth sensor, PrimeSense has not published the technique of depth estimation, some reverse engineering efforts [35] [41] revealed some facts based on which the depth is measured.

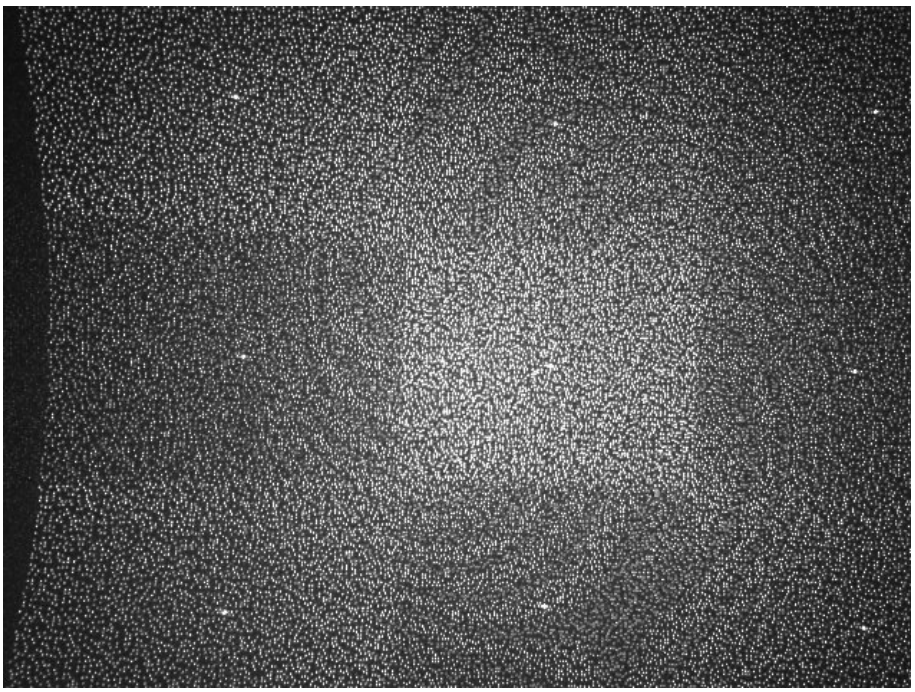


Figure 2.2: Kinect IR pattern projected by the IR projector as a fixed pseudorandom pattern. [35]

In fact PrimeSense is explicitly saying that they are not using time-of-flight, but something they call ”light coding“ and use standard off-the-shelf CMOS sensor which is not capable to extract time of return from modulated light [41]. The IR camera and the IR projector form a stereo pair with a baseline of approximately 7.5 cm [35]. The IR projector can only project a memory-saved fixed pseudorandom pattern, see Figure 2.2.

Stereo triangulation requires two images to get depth of each point (spec) [41]. The technique is to produce one image by reading the output of the IR camera, and the second image is simply the hardwired pattern of specs which laser project. That second image should be hardcoded into chip logic. Those images are not equivalent - there is some distance between laser and sensor, so images correspond to different camera positions, and that allows to use stereo triangulation to calculate each spec depth [41], see Figure 2.3.

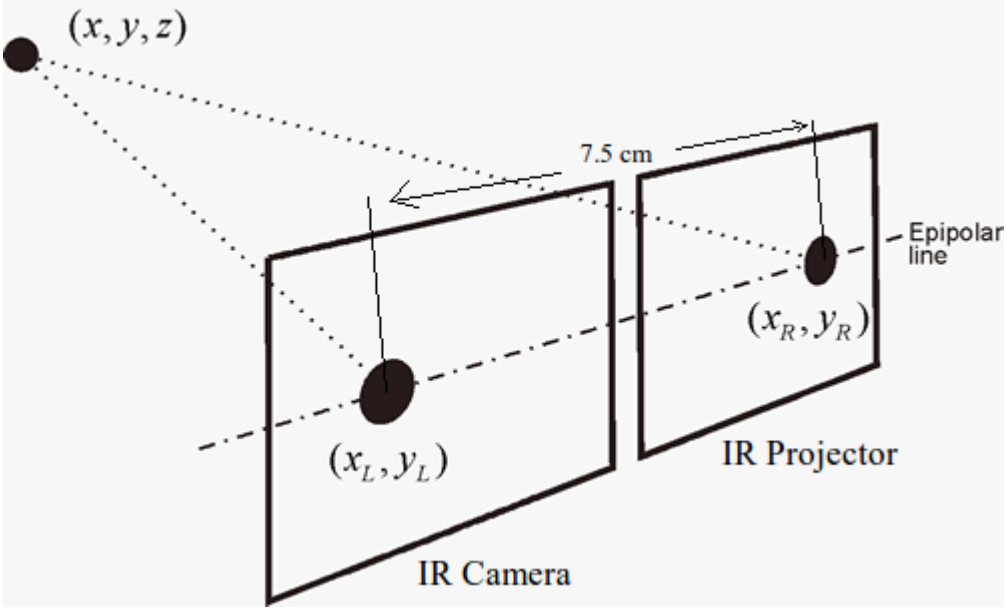


Figure 2.3: The difference here is that the second image in IR projector plane is “virtual” - position of the second point (x_R,y_R) is already hardcoded into memory. Because laser and sensor (IR camera) are aligned the task is even easier: all one has to do is to measure horizontal offset of the spec on the first image relative to hardcoded position (after correcting lens distortion of course). [41]

2.1.2 Lens Distortion

Lens distortion could be one source of increasing the level of inaccuracy in point calculation where any improvement to approach closer to ideal pinhole camera has significant effect on the results. Through non-official efforts [35], calibration of several Kinect devices showed that the typical projection error will vary from 0.34

for originally shipped IR camera to 0.17 pixels after recalibration. The error in comparison with the typical webcams is still small enough even if one relies on the originally calibrated device [35]. In this study, the original calibration was used for developing the algorithms and it was assumed that a calibrated device with intrinsic and extrinsic parameters would be available at the implementation phase.

2.2 Software

Kinect was originally designed to be used in Microsoft XBox 360 gaming console and there is no driver provided by the manufacturer for popular operating systems like Linux at the release time. However, in November 2010, Adafruit Industries [42] offered a bounty for an open-source driver for Kinect. On November 10, Adafruit announced Hector Martin as the winner who had produced a Linux driver that allows the use of both the RGB camera and depth sensitivity functions of the device. It was called OpenKinect's libfreenect which is the core library for accessing the Microsoft Kinect USB camera [34].

Another option introduced by OpenNI organization [44] that provided an open source framework, OpenNI framework, which provides an application programming interface (API) for Natural Interaction devices. Natural Interaction Devices or Natural Interfaces are devices that capture body movements and sounds to allow for a more natural interaction of users with computers in the context of a Natural user interface. The Kinect and Wavi Xtion are examples of such devices [45]. The pupose of developing OpenNI is to shorten the time-to-market of such applications when wanting to port them to use other algorithms, or another sensor [46].

2.2.1 OpenNI framework

The content of this section is based on the provided documentation by OpenNI organization [46]. A highlighted of the framework will be introduced here, for details see the reference documentation.

OpenNI defines "production units", where each such unit can receive data from other such units, and, optionally, producing data that might be used by other units or by the application itself. To support this flow, each such unit is called a Production Node, and all those nodes are connected in a production graph. There are 12 different nodes available listed below:

- **Device** - represents a physical device
- **Depth** - generates depth-maps
- **Image** - generates colored image-maps
- **IR** - generates IR image-maps
- **Audio** - generates an audio stream

- **Gestures** - generates callbacks when specific gestures are identified
- **SceneAnalyzer** - analyzes a scene (separates background from foreground, etc.)
- **Hands** - generates callbacks when hand points are created, their positions change, and are destroyed
- **User** - generates a representation of a user in the 3D space.
- **Recorder** - implements a recording of data.
- **Player** - can read data from a recording and play it.
- **Codec** - used for compression and decompression of data in recordings.

The node type Depth was the main interested in this study which generates a depth image that an equivalent point cloud can be calculated from (see chapter 3).

Chapter 3

Environment Modeling

In research on autonomous mobile robots, many studies on localization and navigation tasks have been conducted. These tasks are generally performed using prior knowledge (e.g., a map of the environment) based on visual and position data of its environment [1]. How to model the environment depends on the sensor data type and the purpose of the using the generated map. In general, there are two major approaches to model an environment for robotics applications: continuous geometric mapping and discrete cell-based mapping. Continuous mapping represents the environment more accurately. Much research has been carried out to develop a path planning method based on a grid representation of the environment [2] [3] [4]. Occupancy grid map represents the environment by discrete cells forming a grid. Here, each cell represents a square area of the environment and stores a value that indicates the occupation state for this area. This is usually done by labeling the cells with "unknown", "free" or "occupied" values or with a value that represents the probability of the cell being occupied or not [5].

In contrast, some path planning algorithms have been developed to deal with continuous modeling approach [6] [7] [8]. Another issue to decide for modeling the world of the robot is 2D or 3D modeling. The world we are living in is 3D (in sense of human being) and this fact is a tempting motivation to model the environment in 3 dimensions, but there are reasons that one may avoid it. As the dimensions increases, the amount of required memory and processing power for dealing with real-time modeling increases as well. Another reason is the fact that the robot is often moving in an indoor planar environment. A 2-dimensional modeling is computationally less expensive and yet an adequate model for the robot to safely plan paths.

3.1 Method

In this study, first a point cloud is calculated and then points which have any height between minimum and maximum height of the robot will be mapped into a 2-dimensional continuous space. The other points are removed from the set.

Because of limitation in field of view of the sensor, it is necessary to rotate the sensor (by utilizing a pan/tilt unit for example) such that it can maintain a wider scope of the environment. A 180° scan at the beginning is carried out to build a model from. At each scan, first the points in the field of view of the captured frame are removed and being updated by new points. At the end of scan, a data clustering is performed for segmentation to categorize each 2D point cloud as a separate set. Next step is to find concave-hull of the points to represent each set with a polygon. At the end of algorithm, an smoothing through vertices of each polygon is executed to remove very close and in one straight line points to reduce the complexity of the configuration space and hence the path planner algorithm's execution time. In the following sections the method will be presented in more details.

3.1.1 Full camera model

As we saw earlier in chapter 2, Kinect depth sensor is indeed an ordinary camera equipped with an IR filter which captures the pattern projected on the scene by the IR projector. It can be model as a pinhole camera at the first step for simplicity of calculations. The pinhole camera model describes the mathematical relationship between the coordinates of a 3D point and its projection onto the image plane of an ideal pinhole camera, where the camera aperture is described as a point and no lenses are used to focus light [28]. A full camera model includes the following transformations [9]:

- The **rigid body motion** between the camera and the scene.
- **Perspective projection** onto the image plane.
- **CCD imaging** - the geometry of the CCD array (the size and shape of the pixels) and its position with respect to the optical axis.

Rigid body motion: A rigid body is one which experiences no change of shape when acted upon by forces however large. [27]. For a full camera model, a rigid body motion is the relation between two coordinates attached to the camera and the scene denoted as $\mathbf{P}^c = (X^c, Y^c, Z^c)$ and $\mathbf{P}^w = (X^w, Y^w, Z^w)$ respectively.

Figure 3.1 illustrates the rigid body motion that can be described by a rotation matrix R and a translation vector T,

$$\begin{bmatrix} X^c \\ Y^c \\ Z^c \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} X^w \\ Y^w \\ Z^w \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \tag{3.1}$$

or equivalently,

$$\mathbf{P}^c = R\mathbf{P}^w + T \tag{3.2}$$

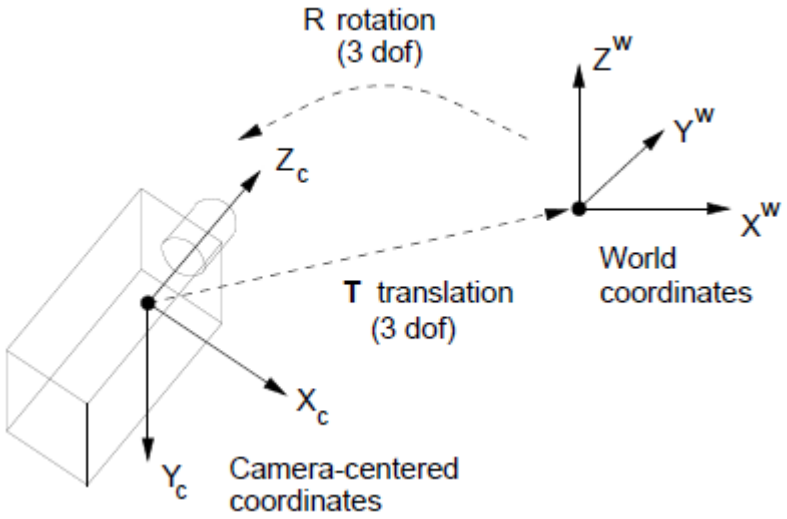


Figure 3.1: Camera rigid body motion can be model by one rotation matrix R and one translation vector T [9].

Perspective projection: The image point in perspective is the point at which a line through the origin (eye) and the world point intersects the image plane [28]. A planar perspective projection onto the imaging surface, utilizing the trigonometry [9], is modeled by (see Figure 3.2),

$$x = \frac{fX_p^c}{Z_p^c} \ , \ y = \frac{fY_p^c}{Z_p^c} \tag{3.3}$$

where (x, y) is the image plane coordinate of the projected point $P = (X_p^w, Y_p^w, Z_p^w)$ onto the image plane. Note that image plane is an imaginary plane which is not discretized at this step.

CCD imaging: A CCD (Charge-coupled device) camera digitalizes the projected scene in the field of view of the camera as a 2-dimensional discrete plane. Figure 3.3 depicts a CCD imaging where we define (discrete) pixel coordinates $\mathbf{q} = (u, v)$ in addition to the image plane coordinates $\mathbf{s} = (x, y)$. If (u_0, v_0) is the corresponding coordinate that origin of $\mathbf{s}$ coordinate is mapped on, and if $1/k_u$ and $1/k_v$ represent the pixel width and height in $\mathbf{s}$ coordinate system, then we can relate the two coordinates $\mathbf{q}$ and $\mathbf{s}$ as,

$$u = u_0 + k_u x \ , \ v = v_0 + k_v y \tag{3.4}$$

Full camera model: As it is mentioned earlier, the simple pinhole camera model does not include any lens distortion as *Radial distortion* and *Decentering distortion* [25]. However, this model is still adequate accurate in absent of a lens distortion

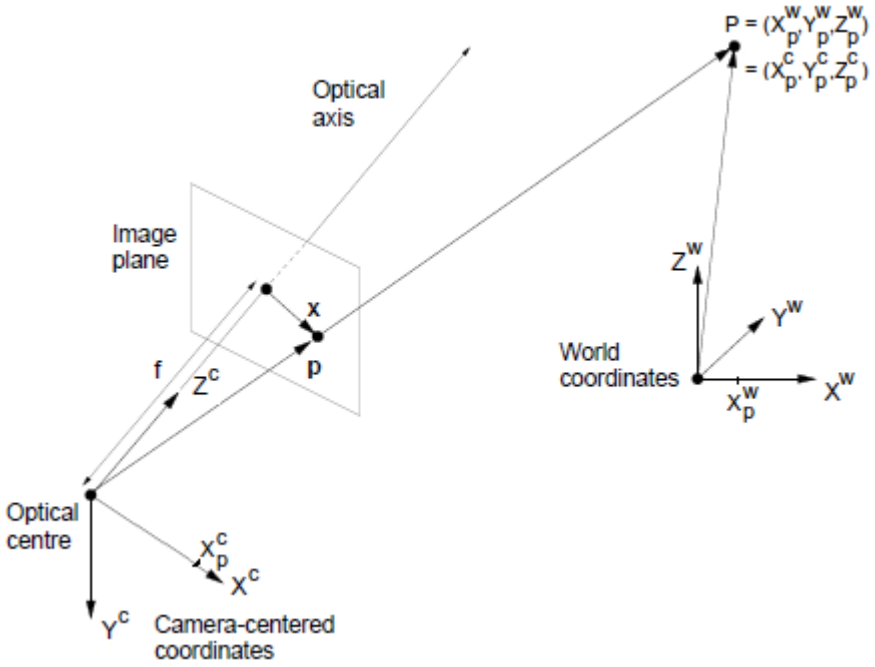


Figure 3.2: Planar perspective projection illustration [9].

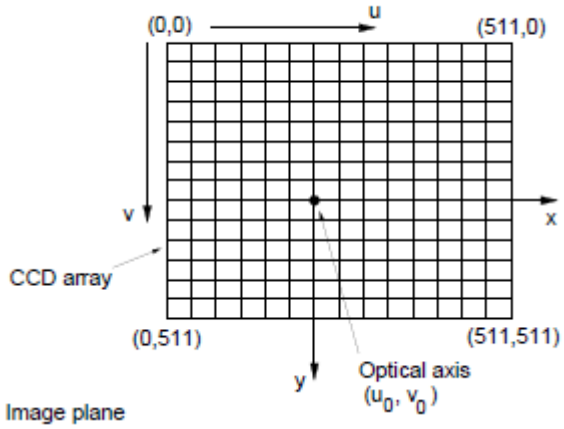


Figure 3.3: CCD-imaging; discrete pixel coordinates $\mathbf{q} = (u, v)$ in addition to the image plane coordinates $\mathbf{s} = (x, y)$ [9].

data as well as the argument regarding the effect of lens distortion in the Kinect sensor in chapter 2. The mapping from the camera coordinate $P^c = (X_p^c, Y_p^c, Z_p^c)$ to pixel coordinate $P^{\mathbf{q}} = (u_p, v_p)$ is maintained by combining (3.3) and (3.4),

$$u_p = u_0 + \frac{k_u f X_p^c}{Z_p^c} \tag{3.5}$$

$$v_p = v_0 + \frac{k_v f Y_p^c}{Z_p^c} \tag{3.6}$$

now fusing (3.1) into (3.7) and (3.8), we obtain the overall mapping from the world coordinate $P^w = (X_p^w, Y_p^w, Z_p^w)$ to pixel coordinate $P^{\mathbf{q}} = (u_p, v_p)$,

$$u_p = u_0 + \frac{k_u f(r_{11}X_p^w + r_{12}Y_p^w + r_{13}Z_p^w + T_x)}{(r_{31}X_p^w + r_{32}Y_p^w + r_{33}Z_p^w + T_z)} \tag{3.7}$$

$$v_p = v_0 + \frac{k_v f(r_{21}X_p^w + r_{22}Y_p^w + r_{23}Z_p^w + T_y)}{(r_{31}X_p^w + r_{32}Y_p^w + r_{33}Z_p^w + T_z)} \tag{3.8}$$

3.1.2 Point cloud

A point cloud is usually defined as a set of unorganized, irregular points in 3D. For example this could be laser range data obtained for the purpose of computer modeling of a complicated ED object [26]. To generate point cloud from depth image captured by the Kinect sensor, one may use formulas (3.7) and (3.8) by calculating the inverse of mapping, that is, given (u, v, d) - where d is the depth and it is equal to Z^c (according to OpenNI software, see chapter 2) - return the corresponding world point (X^w, Y^w, Z^w),

$$x_u = Z^c \times (\frac{u - u_0}{f k_u}) \tag{3.9}$$

$$y_v = Z^c \times (\frac{v - v_0}{f k_v}) \tag{3.10}$$

$$z_d = Z^c = d \tag{3.11}$$

then,

$$\begin{bmatrix} x_u \\ y_v \\ z_d \end{bmatrix} = R \begin{bmatrix} X^w \\ Y^w \\ Z^w \end{bmatrix} + T \tag{3.12}$$

Solving for P^w,

$$\begin{bmatrix} X^w \\ Y^w \\ Z^w \end{bmatrix} = R^{-1} \begin{bmatrix} x_u - T_x \\ y_v - T_y \\ z_d - T_z \end{bmatrix} \tag{3.13}$$

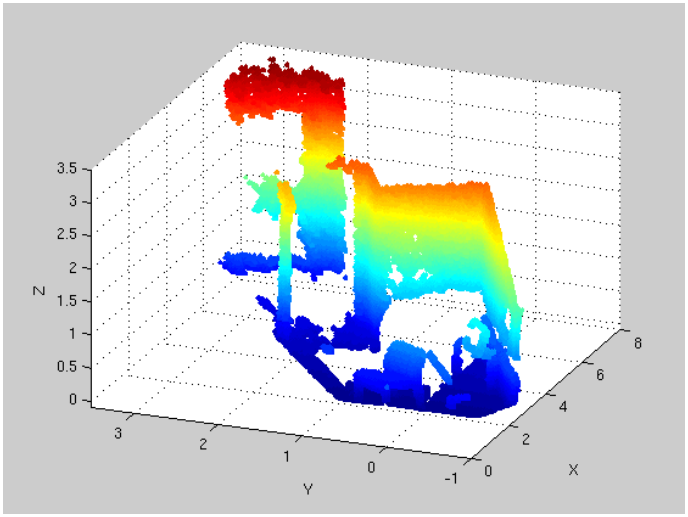
This is the main equation which point cloud is driven from. Note that R and T can be the net of a multiple transformation between coordinates as we will see in

the implementation chapter where there are three coordinates which any point in depth image should be transformed under, to produce the correct point cloud.

An example of generating point clouds from depth image of the Kinect sensor has been illustrated in Figure 3.4.

(a) Image taken from RGB camera.

(b) Depth image driven from OpenNI API.



(c) Point cloud.

Figure 3.4: An example of generated point cloud. Note that each color of points represent a height in (c).

3.1.3 Mapping into 2-dimensional space

The next step is to map the point cloud into 2-dimensional space. To do this, first the points with height between the minimum and maximum height of the robot are retained and all the remain points which cannot be accounted as obstacles will be removed,

$$\mathbf{P}_{2D} = \{(x, y)|h_{min} < z < h_{max}\} \tag{3.14}$$

Figure 3.5 depicts mapping the point cloud in Figure 3.4c into 2D space after removing the floor. Indeed, the points in the figure are the result of the set (3.14).

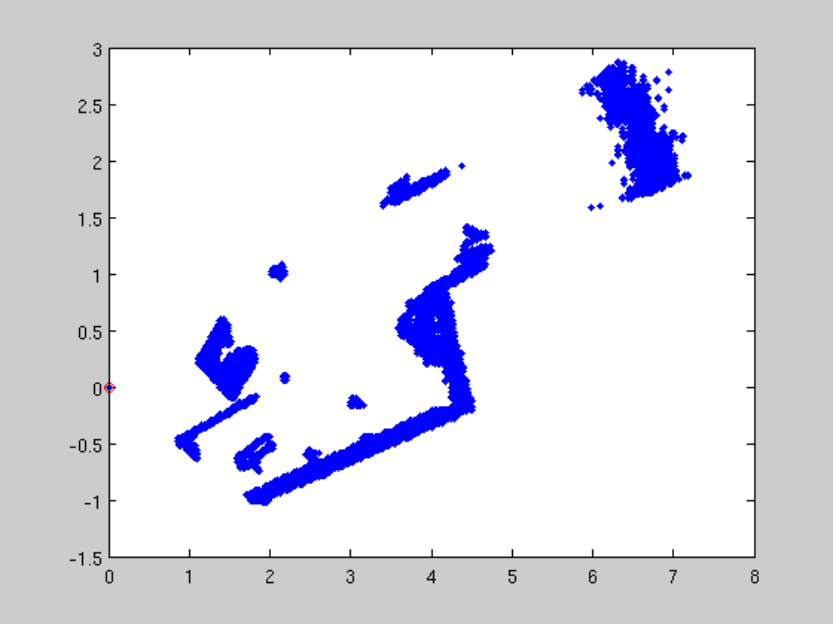


Figure 3.5: The mapped point cloud of Figure 3.4c into 2D space. The camera position is shown by red circle.

It can be seen from the figure that points represent the wall on the other side of the corridor (dense points at the right side of the picture) are distributed such that there is an object with half of meter width! The reason was given in chapter 2 where it was discussed the practical usable range of sensor is upto 3.5 meters. This assumption was widely used in the implementation to obtain more accurate results and avoid false-positive results.

3.1.4 Data clustering

Data clustering (or just clustering), also called cluster analysis, segmentation analysis, taxonomy analysis, or unsupervised classification, is a method of creating groups of objects, or clusters, in such a way that objects in one cluster are very similar and objects in different clusters are quite distinct [17]. The 2D point cloud in the preceding section will need an effective data clustering method to be applied. In this study, we tried Affinity Propagation Clustering (APC), k-mean clustering and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) and the best result was returned by the former method which is very flexible with unusual shape of dense points areas and is able to mark noises as outliers.

Affinity Propagation Clustering: This is a new algorithm recently proposed by Frey in *Science*, which is a kind of clustering algorithm that works by finding a set of exemplars in the data assigning other data points to the exemplars [16].

Unfortunately implementation of this method is not memory efficient especially for large data sets. At the beginning of the method it creates N^2-N similarity pairs where N is the number of data to be clustered. Because we are dealing with a large number of points, this method is slow and computationally expensive. Actually, we couldn't apply it to a real data set because of the reasons mentioned, but it works for small set of data ($N < 400$) quite well.

k-mean clustering: The classic k-means algorithm was introduced by Hartigan (1975; see also Hartigan and Wong, 1978). Its basic operation is simple: given a fixed number (k) of clusters, assign observations to those clusters so that the means across clusters (for all variables) are as different from each other as possible. The difference between observations is measured in terms of one of several distance measures, which commonly include Euclidean, Squared Euclidean, City-Block, and Chebychev [29]. This is one of the simplest data clustering methods which in some applications has been proved to work properly. However, the main problem of using it to cluster 2D-mapped data is that we don't have any prior knowledge that how many dense points areas exist in the data set. In other word, the number of clusters k is unknown and cannot be fixed for any data set. Although there are some adaptive k-mean algorithms [15] in which k is being estimated, but even if we know the number of k the result clustering is not proper for fitting polygons. To see this, an example of k-mean clustering on the data set of Figure 3.5 was performed with different k and the results are shown in Figure 3.6.

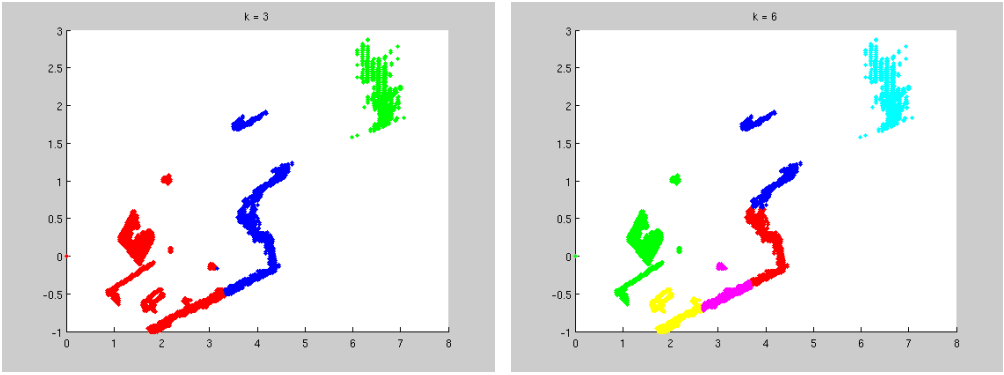
As it depicts, even by increasing the number of clusters, k, we still get a poor clustering where points along a dense line are divided into three or more clusters which is not necessary. Also, in some areas, two or more separated dense points are clustered to the same cluster which means a polygon would block the free space between them and hence reduce the mobility space of the robot. Indeed, these defects are due to the mean based behavior in k-mean algorithm.

Density-Based Spatial Clustering of Applications with Noise: The density-based clustering approach is a methodology that is capable of finding arbitrarily shaped clusters, where clusters are defined as dense regions separated by low-density regions [17]. The following description of the DBSCAN algorithm is based on "Data Clustering" by Martin et al. [17].

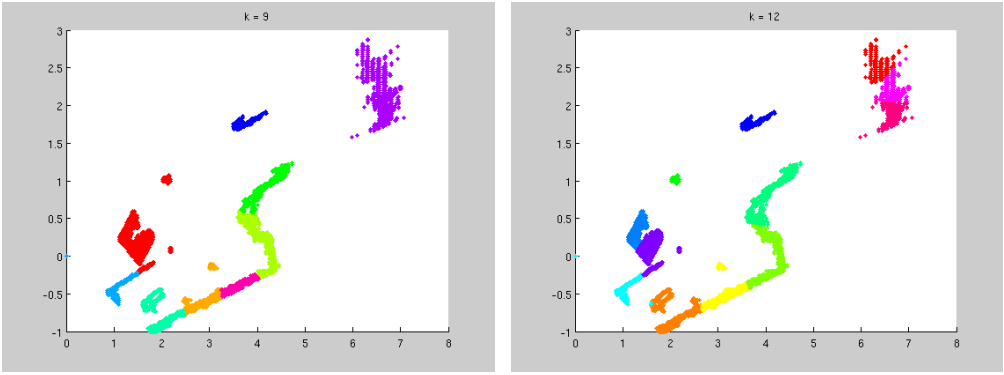
We define an area as the zone for point p in the data set $\mathbf{P}_{2D}$, equation (3.14), which is called **ϵ-neighborhood** of the point,

$$N_\epsilon(p) = \{q \in \mathbf{P}_{2D} | d(p,q) < \epsilon\} \tag{3.15}$$

where $d(p,q)$ is the Euclidean distance between two points.



(a) k = 3 (b) k = 6



(c) k = 9 (d) k = 12

Figure 3.6: Data clustering by k-mean algorithm for different number of clusters k. Even by increasing k, the data clustering is poor due to mean based behavior of the k-mean algorithm.

A point p_i is said to be **directly density-reachable** from another point p_j if p_i be in the zone of p_j and the number of points in the zone be greater than a threshold $N_{\min}$. If there is a sequence of points $\{p_i, p_{i+1}, \ldots, p_{j-1}, p_j\}$ such that each point be directly density-reachable from the next point in the sequence, hence it is said to that point p_i is **density-reachable** from the point p_j.

Two points p_i and p_j are said to density-connected with respect to ϵ and $N_{\min}$ if there exist a point p_k such that both p_i and p_j are density-reachable from p_k with respect to ϵ and $N_{\min}$.

A **cluster** C with respect to ϵ and $N_{\min}$ is a non-empty subset of $\mathbf{P}_{2D}$ satisfying the following conditions:

1. (**maximality**). $\forall p, q \in \mathbf{P}_{2D}$, if $p \in C$ and q is density-reachable from p with respect to ϵ and $N_{\min}$, then $q \in C$.

2. (**connectivity**). $\forall p, q \in C$, p and q are density-connected with respect to ϵ

and $N_{\min}$.

DBSCAN requires two parameters ϵ and $N_{\min}$ as inputs but it generates the best number of clustering automatically. To estimate $N_{\min}$, one can use the heuristic *sorted k-dist graph* since for $k > 4$ does not significantly differ from the 4-dist graph, $N_{\min}$ is set to 4 for all two-dimensional data [17].

We used an implementation of the algorithm by Michal Daszykowski [30] in Matlab (`dbscan.m`) where it statistically calculates ϵ from the given data set $\mathbf{P}_{2D}$. This estimation turned out to work quite well for our data. The parameter ϵ is estimated (in the function `dbscan.m`) as follows,

$$\epsilon = \left(\frac{\lambda(\mathbf{P}_{2D}) N_{\min} \Gamma(\frac{n}{2} + 1)}{m\sqrt{\pi^n}} \right)^{\frac{1}{n}} \tag{3.16}$$

where, m is the number of data, n dimension of data which is 2 for our data set, $\Gamma(.)$ is the gamma function and $\lambda(.)$ is defined as,

$$\lambda(\mathbf{P}_{2D}) = \prod_{i=1}^{m} (\max(x_i, y_i) - \min(x_i, y_i)) \tag{3.17}$$

There was no argument regarding this estimator in the original reference. Therefore, the optimality of the estimator is an open question. However, it seems to be at least a good initialization for the algorithm to obtain acceptable results.

Figure 3.7 illustrates the same data set used for clustering by the k-mean algorithm, was clustered by applying DBSCAN with $N_{\min} = 4$ and ϵ as defined by equation (3.16). However, we found out with $N_{\min} = 10$ we obtain less clusters and more fidelity to the shapes of dense points.

The figure depicts that the resulting clusters contain points which are connected together as if they represent one shape in the space. Comparing this figure with the results shown in Figure 3.6 reveals that not only does DBSCAN not require prior knowledge of k, it even clusters the data in a way that can be approximated by polygons more efficiently. Nevertheless, observing the points in the right most of the figure illustrates the fact that discontinuity in data is the main source for the algorithm to generate many small clusters. Fortunately, these areas are far from the Kinect sensor position and thus the data at that region of space is not accurate enough to retain for data clustering.

DBSCAN data clustering has many benefits, but it is necessary to mention its disadvantages as well. The algorithm utilizes euclidean distance which for any algorithm suffers from the so-called *curse-of-dimensionality*. DBSCAN cannot cluster data sets well with large differences in densities, since the $N_{\min}$ and ϵ combination cannot be chosen appropriately for all clusters then. The time complexity of the algorithm depends on indexing structure of the data. With such an indexing the overall runtime complexity is $O(\log m)$ and without it, $O(m^2)$ where m is the number of data in the set.

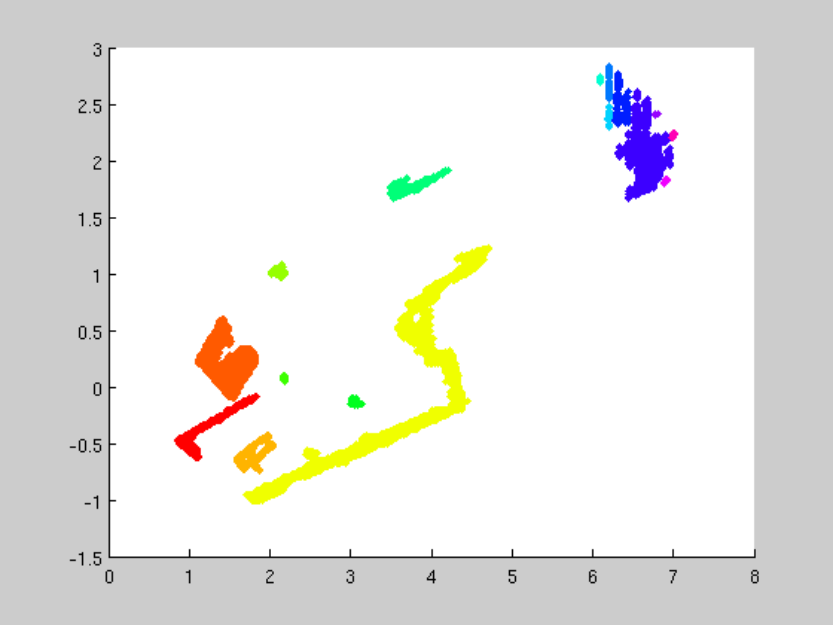


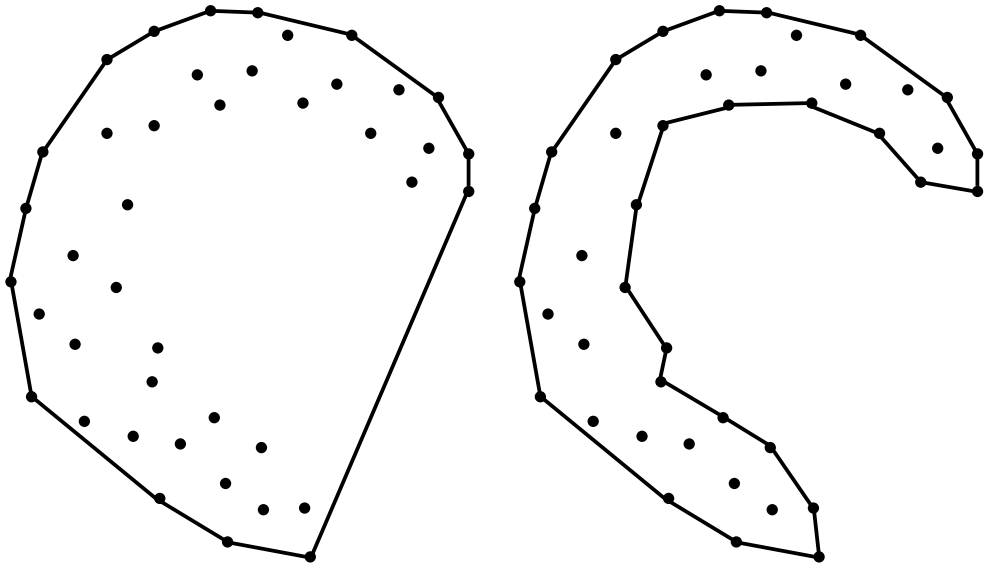
Figure 3.7: Data clustering by utilizing DBSCAN algorithm with $N = 4$ and estimated $\epsilon = 0.0479$. It keeps the connectivity between points and performs clustering of dense points with any shape as one cluster which is very useful to be approximated by a polygon.

3.1.5 Concave-hull estimator

First we define some preliminaries in convex geometry. A convex set is a set in a real vector space V such that for any two points x, y the line segment $[x, y]$ is also containing in the set. The convex-hull of a given set of data points X in V is the intersection of all convex sets in V which contain X [24]. For a given discrete data points, convex-hull can be uniquely defined: find the polygon that its vertices are a subset of data points and *maximize the area while minimizing the perimeter*. In contrast, concave-hull definition will not determine a unique polygon: find a polygon that its vertices are a subset of data points and *minimize the area and perimeter simultaneously*. It is obvious that minimizing both area and perimeter at the same time is a conflicting objective and hence there can be defined many different concave-hull for a given data point. However, we are interested in so-called the best perceived shape (in sense of human being). Some research [14] offers *The Pareto front* which is the south-western frontier in area-perimeter space as the optimum point for minimizing and thus obtaining a polygon which is the same as or very similar to the best perceived shape.

Figure 3.8 depicts the difference between the convex-hull and a concave-hull for the given set of 2-dimensional points.

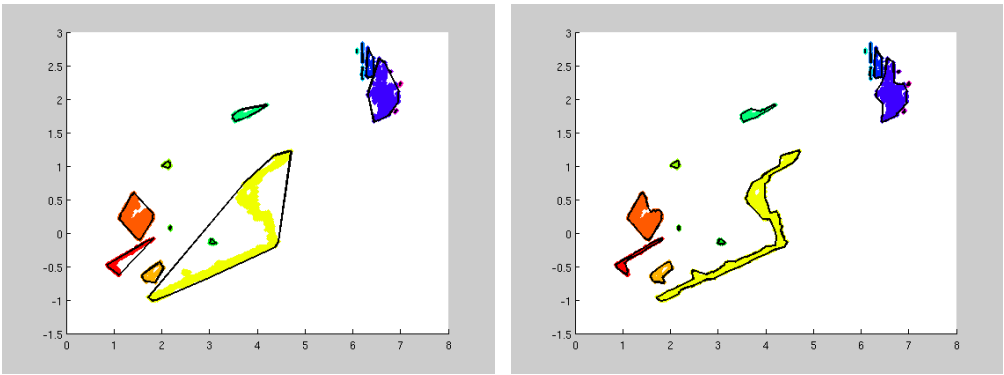
For each cluster resulted from DBSCAN first we apply convex-hull algorithm to



(a) Convex-hull. (b) Concave-hull.

Figure 3.8: The difference between the convex-hull and a concave-hull for the given set of 2D points.

find vertices points. To do this, standard Matlab function `convhull` was utilized to return the set of the convex polygon's vertices in either CCW or CW order. Figure 3.9a shows the result of applying the function `convhull` to each data cluster presented in Figure 3.7 to obtain corresponding convex-hulls.



(a) Applying convex-hull algorithm. (b) Applying concave-hull algorithm.

Figure 3.9: The result of applying convex-hull and concave-hull algorithms to represents obstacles with the best polygons to fit.

The algorithm to calculate a concave-hull of a data set is based on the idea presented by Peter Wasmeier [36] who implemented a function in Matlab called `hullfit`. First, it runs the function `convhull` to obtain vertices of corresponding convex-hull and sort them as clockwise. Then, it performs a distance calculation among all the neighbors' vertices to find any distance greater than maximum allowed distance. If it finds any, then it uses the following two conditions to find another proper point in the data set as the next vertex:

1. Because we know that hull definition is clockwise, the next vertex candidate is the point having the smallest positive angle with the hull line to intersect.

2. The distance to a vertex candidate must be smaller than the length of the line to intersect.

Figure 3.9b depicts the result of applying the function `hullfit`. A significant difference can be seen where the convex-hulls blocks considerably large free spaces whereas the concave-hulls represent the environment with more free space and appropriate fitness of polygons.

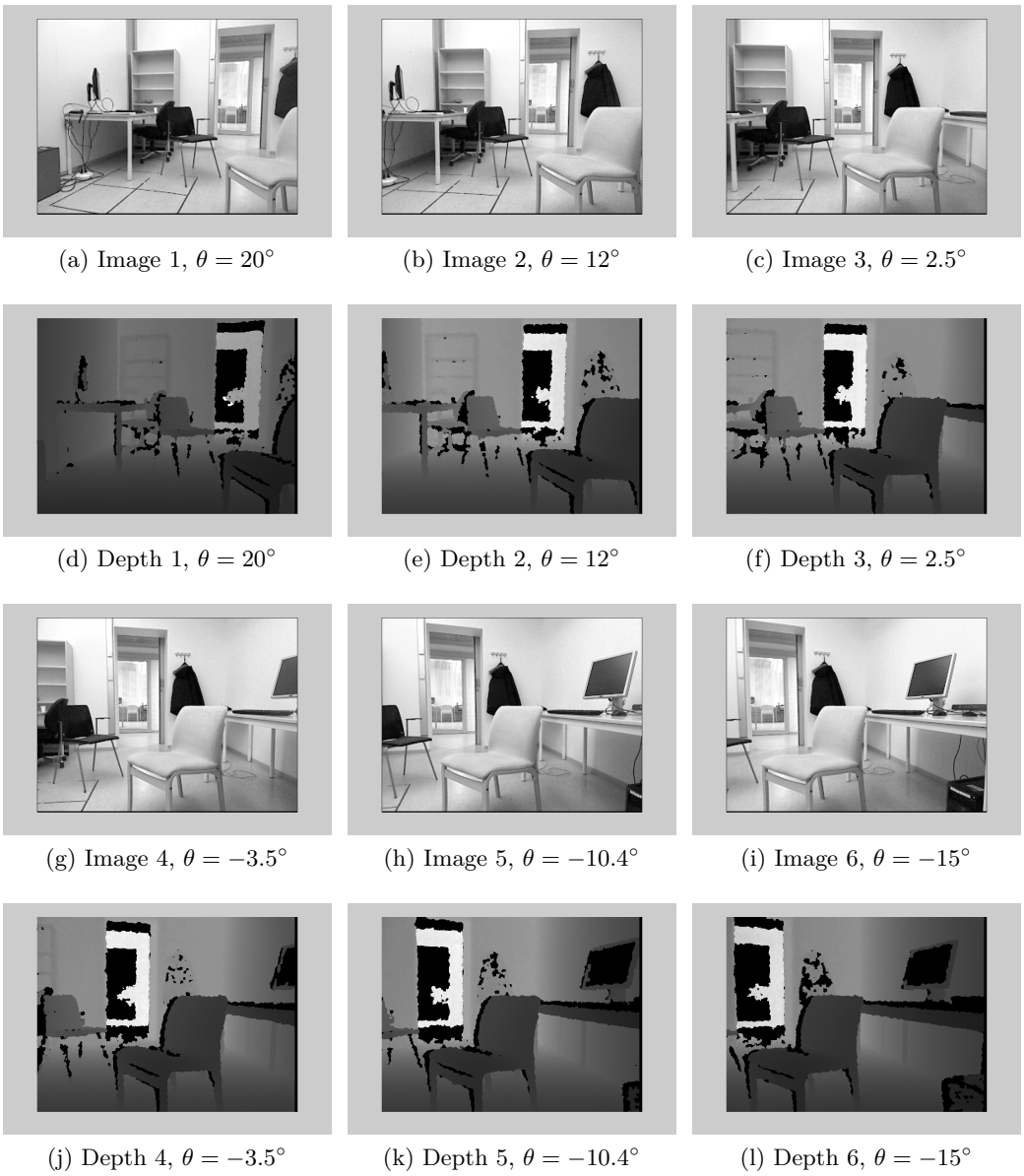
The major disadvantage of applying such algorithm is that we will obtain too many vertices, many of them along a line or approximately can be represent by a line. Therefore, by moving from the first vertex towards the last one and removing such lined the vertices, we perform a smoothing to obtain a reduced number of vertices set.

3.1.6 Building a model of the environment

The vertical and horizontal field of view of the Kinect sensor are 43° and 58° respectively, see chapter 2. Thus, to build a model of the environment around the robot, a mechanism to enable the robot to see a wider field of view seems to be necessary. To do this, one solution we offer is to rotate the sensor such that more frames of depth images can be captured in different angles and then combining these frames to maintain a wider view of the environment.

To illustrate how this works, a set of 6 frames with different orientation angels had been taken by the Kinect sensor clockwise from the left to right of the room. Figure 3.10 shows the images and their corresponding depth image data together. In the scene, there are two chairs placed one behind the other such that a path between them towards the door is available. There are two desks with screen monitors laying on the left and right walls, a bookcase next to the door and a dress hanging with a coat is hanging on at the other side next to the door.

Obviously there is a common field of view between two frames in raw. The points left out of the current field of view will be retained and all the points in the current filed of view first deleted and then updated by the current frame data. Figure 3.11 depicts the process where as the more frames are captured, the more points are included in the 2D-mapped space. The simplest implementation would be adding the new frames' 2D-mapped points to the world subset $\mathbf{S}_w$. This approach imposes



(a) Image 1, $\theta = 20°$ (b) Image 2, $\theta = 12°$ (c) Image 3, $\theta = 2.5°$

(d) Depth 1, $\theta = 20°$ (e) Depth 2, $\theta = 12°$ (f) Depth 3, $\theta = 2.5°$

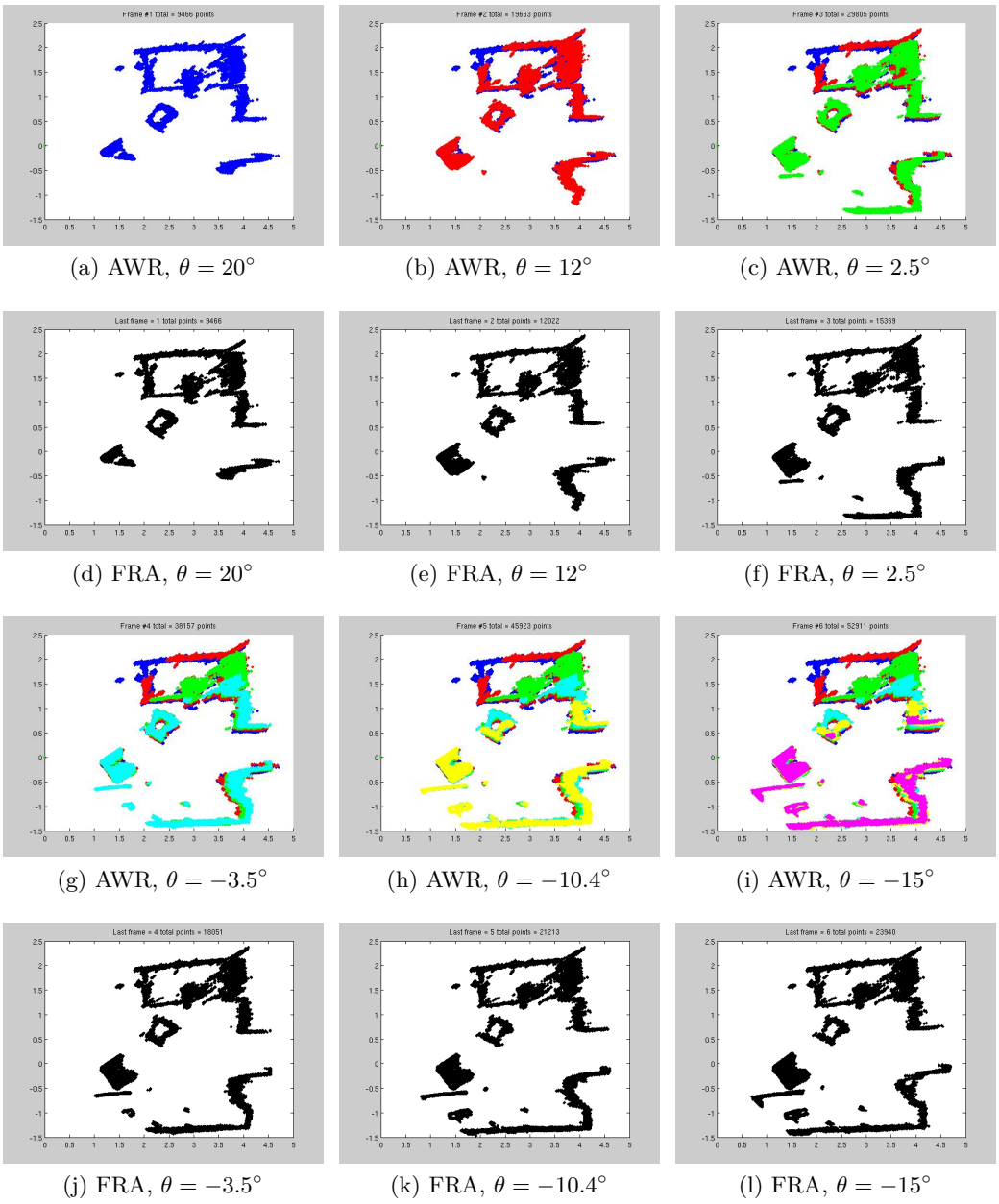
(g) Image 4, $\theta = -3.5°$ (h) Image 5, $\theta = -10.4°$ (i) Image 6, $\theta = -15°$

(j) Depth 4, $\theta = -3.5°$ (k) Depth 5, $\theta = -10.4°$ (l) Depth 6, $\theta = -15°$

Figure 3.10: A sequence of frames taken while the Kinect sensor was sweeping the environment from left to right of the room clockwise to obtain a wider view.

two problems; first, according to time complexity of the data cluster DBSCAN, as the number of points increases, the time of clustering increases, too.

Very many points will be located very close to each other which do not contribute to build a more accurate world. Another problem arises from the fact that in the real world there exist dynamic obstacles which should be updated for each new



(a) AWR, $\theta = 20°$ (b) AWR, $\theta = 12°$ (c) AWR, $\theta = 2.5°$

(d) FRA, $\theta = 20°$ (e) FRA, $\theta = 12°$ (f) FRA, $\theta = 2.5°$

(g) AWR, $\theta = -3.5°$ (h) AWR, $\theta = -10.4°$ (i) AWR, $\theta = -15°$

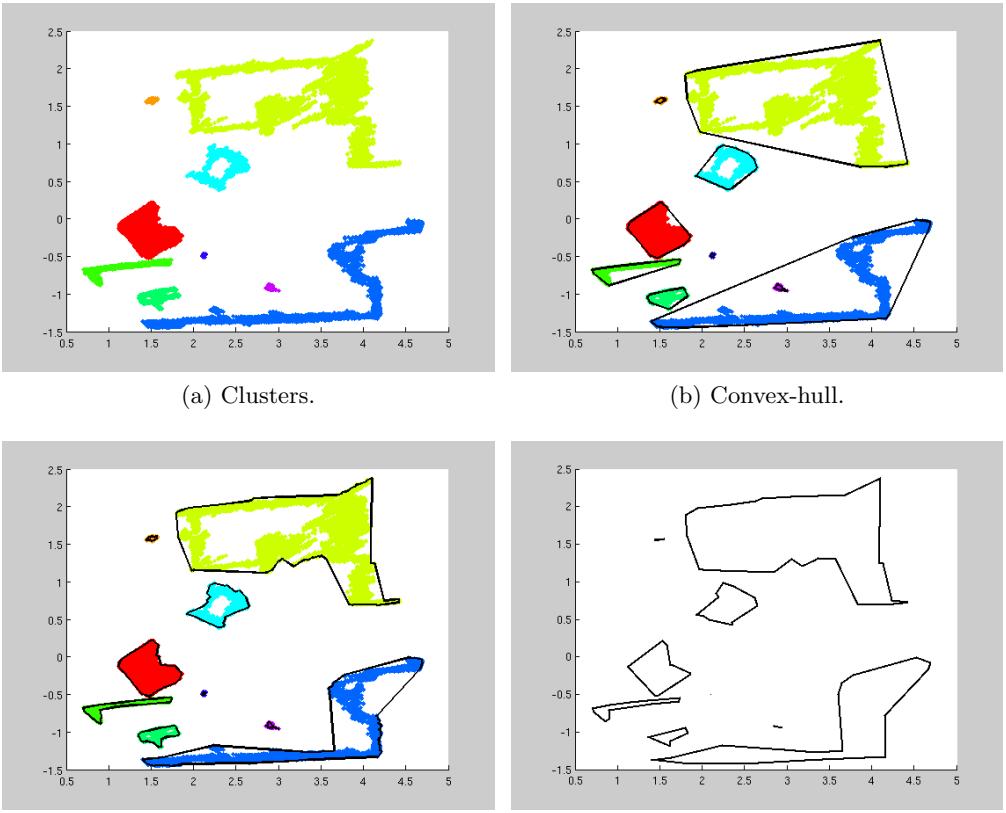
(j) FRA, $\theta = -3.5°$ (k) FRA, $\theta = -10.4°$ (l) FRA, $\theta = -15°$

Figure 3.11: The equivalent sequence of 2D-mapped depth frames. Notice the significant difference between the number of points remain in Adding Without Removing (AWR) the pints in the current field of view, $n = 52911$ and First Removing those points then Adding new points (FRA), $n = 23940$.

frame. We name the first simple method as Adding Without Removing (AWR) the points in the current field of view. The second method is one solution of the

problems mentioned above. We call this approach as first removing (the point in the current field of view) then adding new points (FRA). These are illustrated in Figure 3.11 where the significant difference between the number of remained points at the end of each method can be seen: $n_{\text{AWR}} = 52911$ versus $n_{\text{FRA}} = 23940$.

The next step is to cluster the points in the set $\mathbf{S}_w$, finding the concave-hull of each cluster and applying a smoothing algorithm to reduce the number of vertices for each polygon. These steps are depicted in Figure 3.12.



(a) Clusters.

(b) Convex-hull.

(c) Concave-hull.

(d) Smoothing.

Figure 3.12: Steps to find the final model of the environment. (a) DBSCAN data clustering. (b) Applying convex-hull on each cluster. (c) Applying `hullfit` to approximately find concave-hull of each cluster. (c) Applying the smoothing algorithm to reduce the number of vertices, final model of the environment.

Chapter 4

Obstacle Avoidance and Path Planning

Within the mobile robotics research community, a great many approaches have been proposed for solving the navigation problem [18]. Navigation remains one of the most challenging functions to perform, in part because it involves practically everything about AI robotics: sensing, acting, planning, architectures, hardware, computational efficiencies, and problem solving [19]. Given a map and a goal location, path planning involves identifying a trajectory that will cause the robot to reach the goal location when executed. In comparison, given real-time sensor readings, obstacle avoidance means modulating the trajectory of the robot in order to avoid collisions [18].

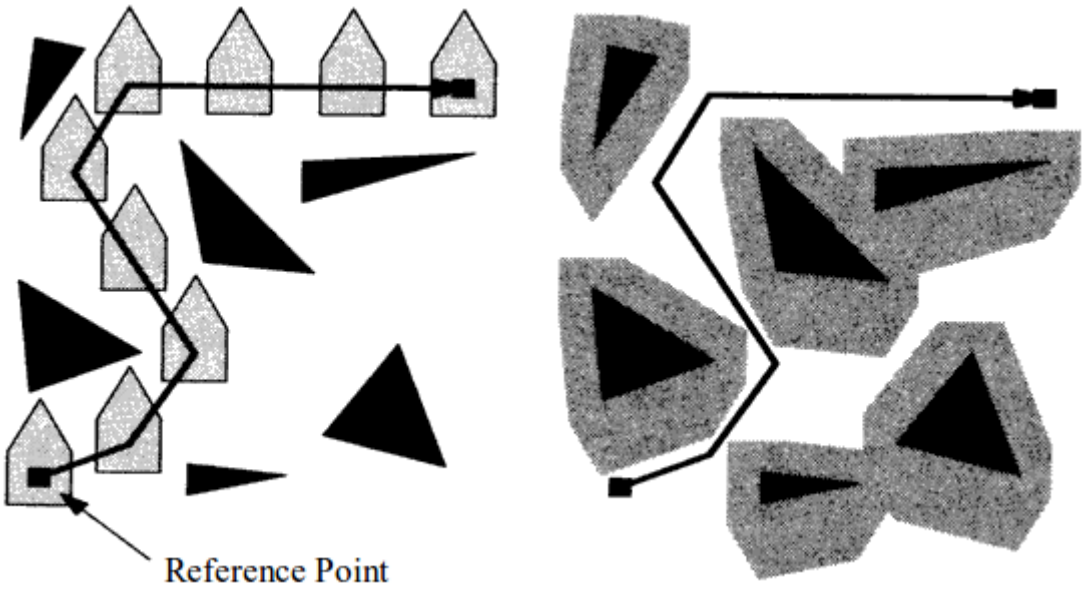
Perception is the key in method selection for obstacle avoidance. Over the years, a broad research on developing algorithms to tackle the problem has been performed. Some of which are specifically suited to working with 2D laser scanners and sonars [11] [12] while some other approaches can be fit to other sensors. In comparison, main focus of some other works was on developing new methods or adapting the methods borrowed from 2D solutions for 3D perception usually utilized by vision sensors in, for example, stereo vision configuration or time-of-flight cameras [10].

The robot's environment representation can range from a continuous geometric description to a decomposition-based geometric map or even a topological map [18]. There are strategies to make all the representations practically feasible. However, each strategy has its own advantages and weak points. Topological maps in where a set of connected navigation nodes represent the world is mostly used for global path planning. Such global planning is beyond the scope of this study.

4.1 Configuration Space

The first step in finding a path to a goal is usually to transform the space in both continuous geometric or decomposition-based form, into a representation in which the robot will be seen as 2D (or 3D) point and the obstacles will be grown up to the shape of the robot properly. This transformed representation is called configuration

space or briefly C-Space. This step is essential to simplify significantly the method of path planning and obstacle avoidance. Figure 4.1a shows an example of working-space in which there is a robot indicated with gray color assumed to be holonomic and a set of obstacles. In berif, to generate C-Space, first select a point inside or on the border of the robot's shape (the polygon describing the shape of the robot) and call it as reference point. Then move the robot polygon all around any obstacle in working space without any collision (contacting of borders of obstacles and the robot is permitted). By moving around each obstacles as mentioned above, the reference point will draw a curve around each obstacle. These new curves determine the border of the obstacles in C-Space where the robot is represented as the reference point. This new configuration of space is the generated C-Space (see Figure 4.1b).



(a) Working-Space (W-Space) (b) Generated Configuration Space (C-Space)

Figure 4.1: An example shows how to generate a configuration space from working space. (a) The robot is assumed to be holonomic. (b) Obstacles are grown up while the robot is presenting as a point.

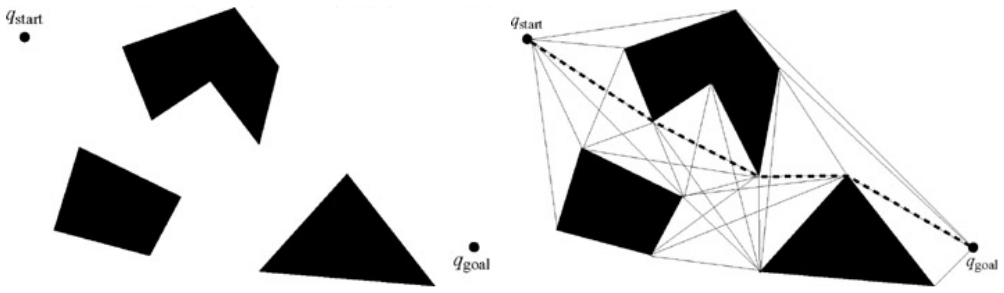
4.2 Continuous Geometric Environments

In this representation, the environment is a continuous 2D (or 3D) area (or volume) that, depending on the data structure of obstacles, filled with a set of points, line segments, polygons and so forth [13] [23]. A path connecting the current location of the robot in C-Space to the goal pose is usually formed by a network of one-

dimensional curves and/or lines called *road map*. Two approaches for forming a road map are *Visibility graph* and *Voronoi diagram* [18].

4.2.1 Visibility graph

The standard visibility graph is defined in a two-dimensional polygonal configuration space (Figure 4.2a). The nodes of the visibility graph include the start location, the goal location, and all the vertices of the configuration space obstacles. The graph edges are straight-line segments that connect two line-of-sight nodes [20]. An optimum path starting from the start pose to the goal can be found by searching on the graph utilizing a standard optimum search method such as A-Star search algorithm [19]. More formally, we can prove that visibility graph planning is optimal in terms of the length of the solution path. This powerful result also means that all sense of safety, in terms of staying a reasonable distance from obstacles, is sacrificed for this optimality [18]. Figure 4.2b shows the result of applying the method by highlighting the path as a dotted line connecting start pose to the goal.



(a) Polygonal configuration space with a start and goal.

(b) The shortest path in thick dotted line.

Figure 4.2: An example of path planning using **Visibility graph** method. [20]

4.2.2 Voronoi diagram

Voronoi diagram in basic form is a special decomposition of metric space where each point on the diagram has an equidistant from two or more points given as a set in the space. The generalized Voronoi diagram (GVD) is the set of points where the distance to the two closest obstacles is the same [20]. For each point in the free space, computing its distance to the nearest obstacle and plotting that distance as a height results that higher as moving away from an obstacle. At points that are equidistant from two or more obstacles, such a distance plot has sharp ridges. The Voronoi diagram consists of the edges formed by these sharp ridge points [18]. Figure 4.3 shows an example of generating Voronoi diagram for given metric space and finding a path through the diagram connecting the goal to the start pose.

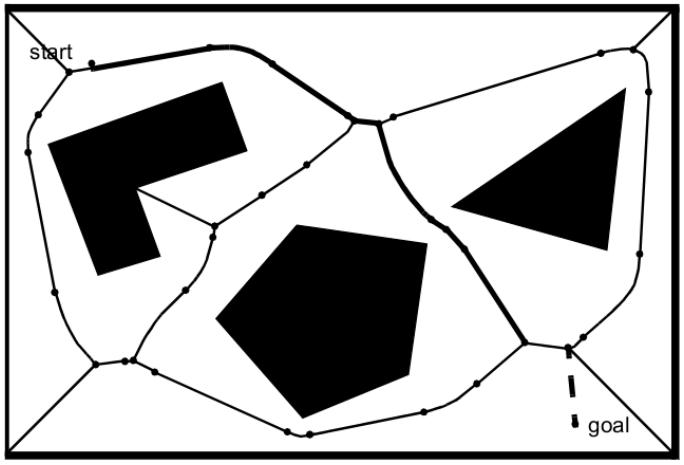


Figure 4.3: An example of generated **Voronoi diagram** for given metric space with some obstacles as polygons. The points on the Voronoi diagram represent transitions from line segments (minimum distance between two lines) to parabolic segments (minimum distance between a line and a point) [18].

The Voronoi diagram has an important weakness in the case of limited range localization sensors. Since this path-planning algorithm maximizes the distance between the robot and objects in the environment, any short-range sensor on the robot will be in danger of failing to sense its surroundings [18].

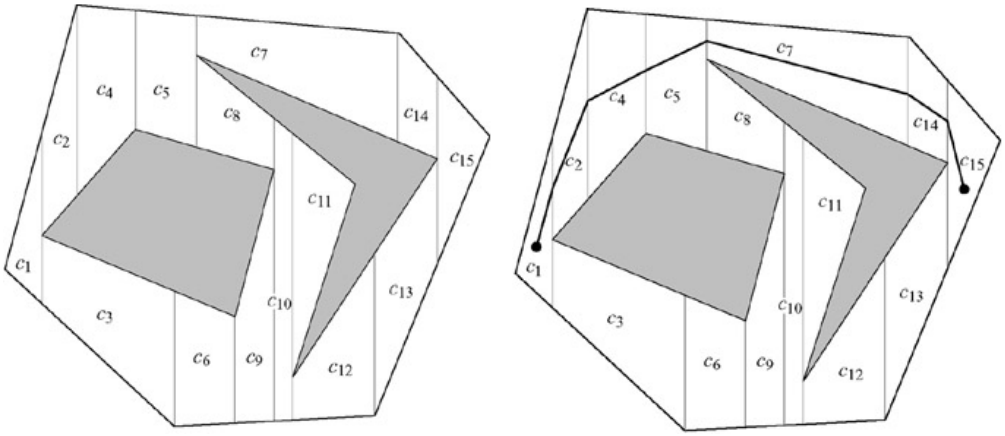
Another shortcoming arises from the fact that the path presenting by a Voronoi diagram although is safe in sense of maximizing the distance between the path and obstacles, but it is obviously suffering from non-optimality of path-length.

4.3 Decomposition-based Environments

Representations in which the environment is divided into many sub-areas usually called cells or grids, decompose the world into free and occupied areas. There are two basic approaches: Cell decomposition and Occupancy grid mapping.

4.3.1 Cell decomposition

These structures represent the free space by the union of simple regions called cells. The shared boundaries of cells often have a physical meaning such as a change in the closest obstacle or a change in line of sight to surrounding obstacles (see Figure 4.4a) [20]. The path in basic form, will be chosen from the set of middle points of the boundaries such that a connection between start and goal pose can be found (see Figure 4.4b).



(a) cell decomposition of the environment. (b) Path finding through boundaries.

Figure 4.4: An example of path planning using **Cell decomposition** method. [20]

4.3.2 Occupancy grid mapping

One solution to avoid using continuously defined environments is to represent the environment as an array of (finite) cells. Each cell can be treated as filled, empty or unmarked if the corresponding area of the cell in the environment is occupied, free or unknown respectively (see Figure 4.5). Another type of grid map could assign a probability of being free (or occupied) to each cell.

Although griding makes a simpler bounded data to work with, it suffers from sacrificing accuracy to obtain data reduction. It is also necessary to define a limit of the modeled environment. The path planning over a given occupancy grid map is usually performed by a search method on all cells. A-star ensures to find the shortest path connecting cells from the start to the end point if there exist such route in the bounded grid map. The result is constructed by very many small straight line segments which in general form a non-smooth path.

4.4 Method

The method one chooses for path planning is directly related to the answer to the higher level question regarding the modeling of the environment. The path planner developed in this work is based on Visibility graph. One reason for that is because of the continuous geometric nature of representation of the environment described in the preceding chapter. Another and the main reason for selection of Visibility graph is the fact that it ensures to find the shortest path between two points. However, it suffers from the fact that if the robot follows this shortest path, it actually has to turn very closely around the vertices of polygons as well as in some cases move very closely in parallel of polygon's line segments. Hence, the developed algorithm

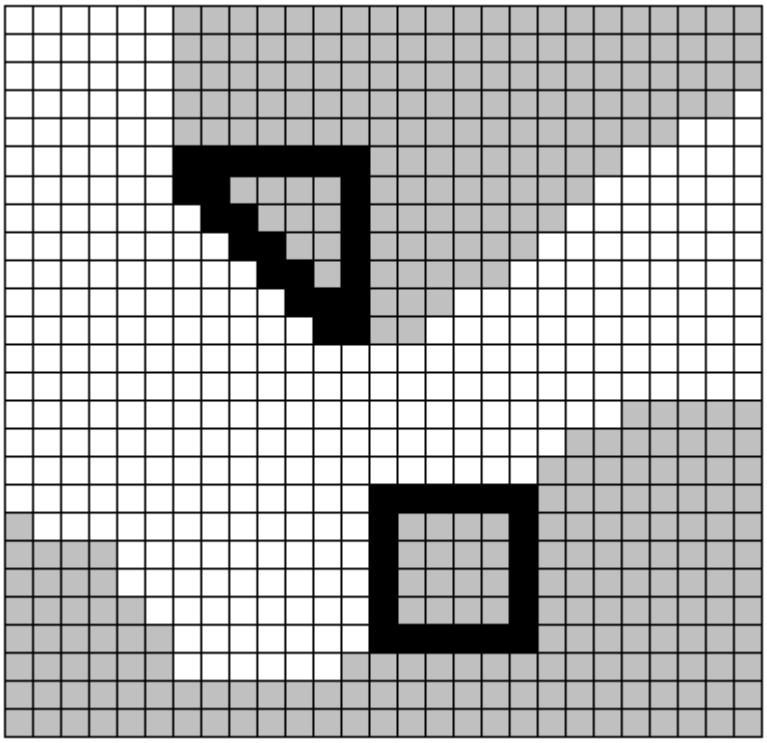


Figure 4.5: A sample **Occupancy Grid** representation of the environment. White cells are free space, black cells are occupied by some obstacle and gray cells are unknown.

is a modification in which it finds safer paths at the cost of reducing the optimality of the distance. In practice it mostly avoids the mentioned problems.

4.4.1 Finding the vertices

The first step in implementation of Visibility graph is to determine vertices of polygons that represent obstacles in the environment. From the preceding chapter, we remember that the obstacles in the environment are represented by a set of line segments such that they form polygons. Thus, the two end points of each line segment are potentially vertices. However, for any polygon, each end point of a line segment is one and only one vertex of the polygon.

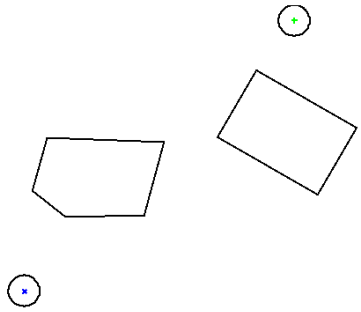
If the vector $\mathbf{l}_i = [x_1, y_1, x_2, y_2]^T$ indicates the i-th line segment by holding the end points coordinates, the set of line segments can be represented as

$$S_l = \{\mathbf{l}_i | i = 1 \dots n\} \tag{4.1}$$

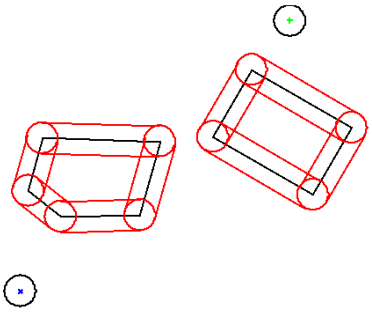
The set of unique vertices then extracted from the set (4.1) such that either the starting points or end points of the vectors $\mathbf{l}_i$ form the vertices set,

$$S_v = \left\{(x_j^{l_i}, y_j^{l_i}) | i = 1 \dots n\right\} \; ; \; j = 1 \text{ or } 2 \tag{4.2}$$

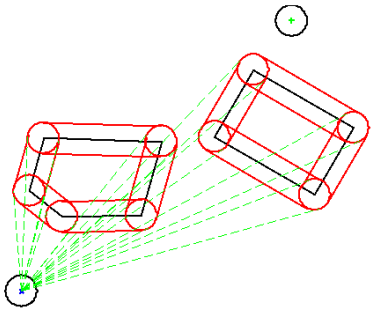
Figure 4.6a shows an example of settings in which two obstacles - sitting between the the robot location (black $\times$) and the goal position (green $+$) - are approximated by two polygons. The equivalent configuration space considering the robot as a circular object with radius r then generated by growing up the polygons. Figure 4.6b shows the configuration space where vertices are grown to a circle with the same radius as the robot. Hence, the problem of finding vertices such that the line connecting the current position of the robot to those vertices without colliding with any obstacle, is transformed to the problem of finding the set of line segments that their tangents to the circle vertices don't intersect with any polygon in the configuration space, see Figures 4.6c and 4.6d.



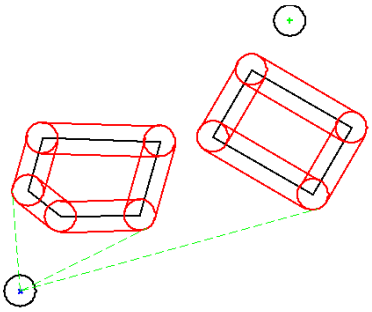
(a) Obstacles as polygons.



(b) The equivalent configuration space.



(c) Tangents to the circle vertices.



(d) After filtering.

Figure 4.6: A sample environment where two obstacles are approximated by two polygons.

4.4.2 Searching for non-colliding path

Observing the three tangent points in Figure 4.6d, one may intuitively perform the step in Figure 4.6c for each point to find the next set of tangent lines and then apply the filtering process to maintain non-colliding subset of circle vertices as the next candidate set. This process is indeed a search through all possible combinations of line segments which connect the current pose of the robot to the goal in forward mode. It is not interested to look back and take into account the vertices are left behind at the current point of the search process while at the same time the search should be complete. Obviously finding the shortest path is another important factor of searching that should be considered. This optimality in the path length and on the other hand the search time, pose the selection of an existing search algorithm as a trade off between these two parameters. In this work, the famous A-star search was utilized for this purpose.

A-star search algorithm: The A-star search algorithm is an extension of Dijkstra's algorithm that tries to reduce the total number of states explored by incorporating a heuristic estimate of the cost to get to the goal from a given state [21]. It utilizes an evaluation function,$f(n)$ of the current node n as the sum of the cost to reach to the node,$g(n)$ and a heuristic function $h(n)$ which is an estimate of the cost left to reach to the goal from the node n,

$$f(n) = g(n) + h(n) \tag{4.3}$$

where,

$$g(n) = \sum_{i=1}^{n} \mathrm{d}(i) \ ; \ \mathrm{d}(i) = \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2} \tag{4.4}$$

and node $i = 0$ is the starting node.

The tree-search version of A-star is optimal if $h(n)$ is admissible, while the the graph-search version is optimal if $h(n)$ is consistent [22]. Although it seems that the visibility graph requires a graph search, but by preventing circuits and self loops in the graph, we reduce it to the tree search problem. Performing a tree search, optimality is maintained by defining a heuristic function such that it never overestimates the cost to reach to the goal [22],

$$h(n) = \sqrt{(x_n - x_0)^2 + (y_n - y_0)^2} \tag{4.5}$$

After calculating the evaluation function $f(n)$, for the cheapest solution, in turn, it is reasonable to try first the node with the minimum value of $f(n)$. An algorithm to implement A-star search is presented in Table 4.1 which the simulation of path planner in Matlab was carried out based on.

A-star algorithm

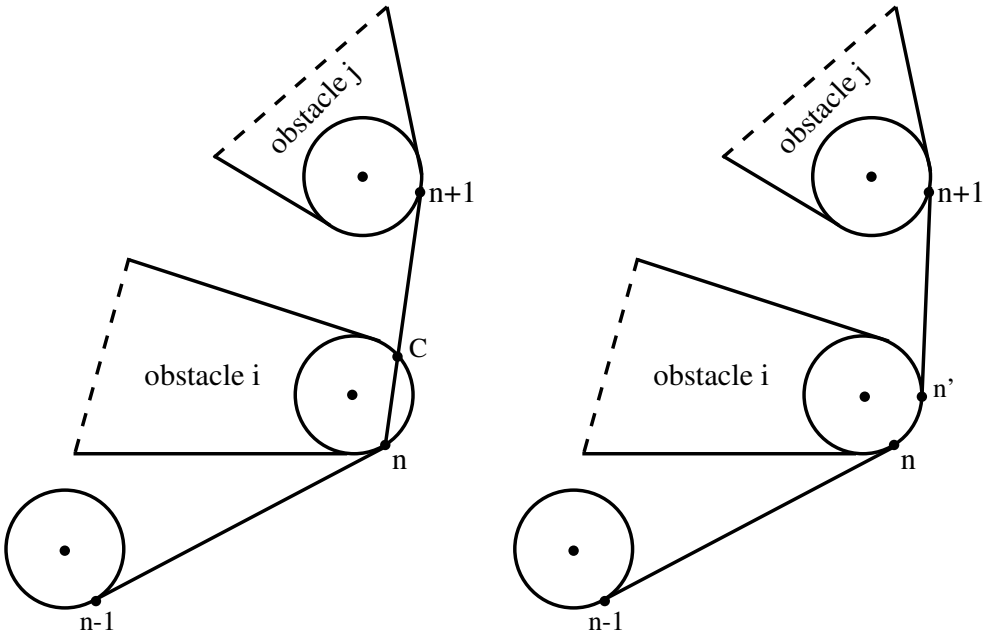
create the open list of nodes, initially containing only our starting node
create the closed list of nodes, initially empty
while(we have *not reached* our goal) {
consider the best node in the open list (the node with the *lowest f value*)
if(this node *is the goal*) {
then we're done, return the solution.
}
else{
move the *current node to the closed list* and consider all of its neighbors
for(each *neighbor*) {
if(this *neighbor is in the closed list* and our current g value is lower) {
update the neighbor with the new, lower, g value
change the neighbor's parent to our current node
}
else if(this *neighbor is in the open list* and our current g value is lower) {
update the neighbor with the new, lower, g value
change the neighbor's parent to our current node
}
else this neighbor is *not in either the open or closed list* {
add the neighbor to the open list and set its g value
}
}
}
}

Table 4.1: Implementation of A-star algorithm in Pseudo-code [47].

4.4.3 Arc joints

As the search process progresses, it adds more points which identify the final path. Because they control the behavior of the path, there is a tend to call them as control points. It is necessary to study the geometric situation in more detail. One situation that imposes the planner to deal with curves joining the line segments is illustrated in Figure 4.7. The line starting from a tangent point of some circle vertex might cross its own circle (see Figure 4.7a) while it connects this point to the next tangent point of another circle vertex. Therefore, instead of using each tangent point as the starting point for the next edge in the graph, one solution is to find common tangent line (segment) of two circle vertices, see Figure 4.7b. This forces to represent the joints of the path with an arc of circle. Thus, the complete path is formed by a set of line segments and circle arcs,

$$\text{path} = \{\mathbf{l}_0, \mathbf{a}_1, \mathbf{l}_1, \mathbf{a}_2, \ldots, \mathbf{l}_n\} \tag{4.6}$$



(a) Conflicts. (b) One solution.

Figure 4.7: (a) A situation in which the line connecting the tangent point n to $n+1$ intersects the circle vertex n at the point C. (b) One solution is to find the common tangent line $\overline{(n+1)(n')}$ and add the curve $\widehat{(n)(n')}$ to the path between two line segments.

where,

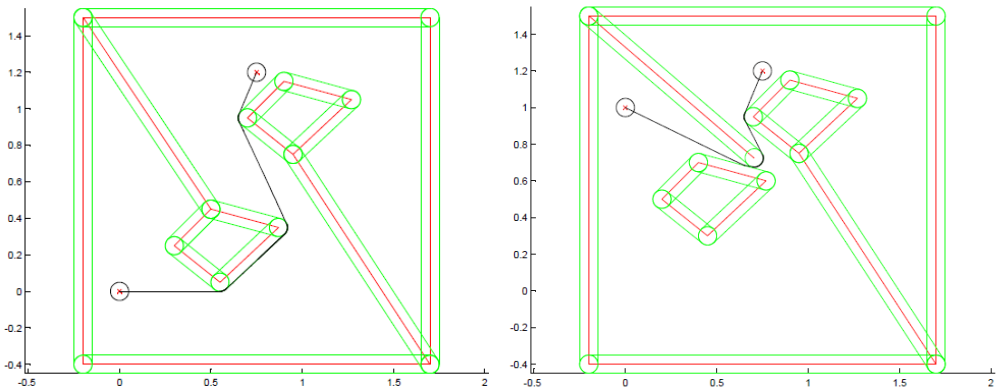
$$\mathbf{a}_i = \{(x_{c_i}, y_{c_i}, r_{c_i}), (x_{s_i}, y_{s_i}), (x_{e_i}, y_{e_i})\} \tag{4.7}$$

is the i-th arc identified by the center and radius of the circle, starting and end points respectively. Although it seems that it is not necessary to include the radius of the circle, because it is fixed to the robot's radius for all the circles in the configuration space, but for the reason will be discussed in the next paragraphs, we retain the radius in the set.

4.4.4 A safeguard for the path

At this point the path planner is able to generate a non-colliding path given the environment with a set of lines/polygons. Figure 4.8 shows some examples in which the paths were computed by the path planner implementation in Matlab. For each given environment the path is the shortest non-colliding route from the starting point to the goal. As it can be seen in sample 1 the robot following these routes has to move very closely to the obstacles to retain the criterion of shortness optimality. In practice, all the uncertainties in measurements, models and control make

it almost impossible for robot to follow such paths without collision. Hence, it is essential to add a safeguard mechanism to the path planner so that it can find safer path in sense of having a reasonable distance from the obstacles, if it is possible. For example, in sample 2 there is no free space between the line and the polygon touching each other in the middle of the environment.



(a) Sample 1. (b) Sample 2.

Figure 4.8: A sample set of path planning result simulated in Matlab. Notice the closeness of paths to obstacles without a safeguard mechanism.

One solution to integrate the safety mechanism is illustrated in Figure 4.9. By growing up the circle vertices to some reasonable radius ($r' > r$), the path finds a distance to the obstacles.

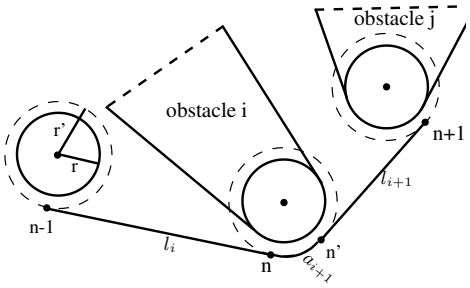
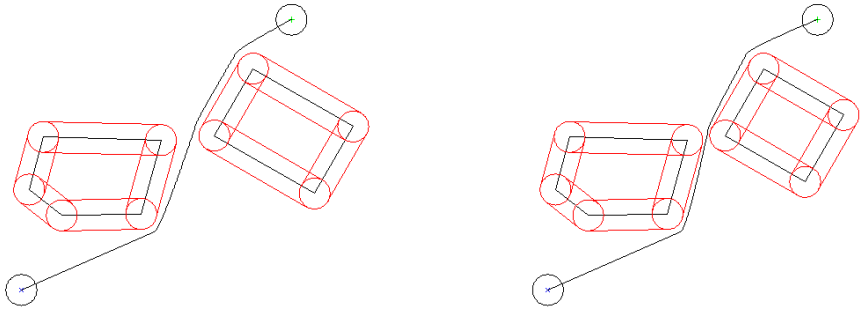


Figure 4.9: One solution to modify path such that it keeps a safe distance from the obstacles.

However, it is not possible for all the cases to grow up the circle vertex to r'. If the maximum of expanding the radius is r', then first it is checked to find out if the grown circle intersects any other obstacle, if so, another radius can be tested. This new test can be carried out by dividing r' by two for a trading off between speed of tests and keeping the maximum available distance from the objects.

Figure 4.10 shows two examples of the final path planner output considering a safeguard in the path.



(a) Sample 1.

(b) Sample 2.

Figure 4.10: The final path planner results for two different scenarios. (a) If there is enough free spaces between obstacles, a safeguard will be maintained. (b) If the obstacles are closer than minimum required distance, then the path planner allows the path to pass through them with the maximum possible safety gap.

In sample 1, because there is enough free space between two obstacles, the path gets a distance from obstacles along the way to the goal. The situation in sample 2 where two obstacles are closer to each other such that there is not enough gap between them, makes the path planner to allow more approaching towards the obstacles.

4.4.5 Obstacle avoidance

After building a model of the environment and generating a pre-computed path according to all the methods described in this and preceding chapter, the robot starts following the path while it is capturing data from the Kinect sensor. This data is continuously converted to a point cloud. If the path is being intersected by any circle with a center of point cloud and the radius of the robot, and the distance between the current pose and the intersection point is less then some threshold, then robot stops, adds the current point cloud to the world and performs all the steps to build a new model of the environment.

Having this new model, the path planner computes for another short but safe path from the current pose of the robot towards the goal. However, the previous path is retained and if during re-planning the obstacle which blocked the path is removed (for example a dynamic obstacle) then the robot will continue the original path. In other words, the dynamic changes in the environment which do not have any influence on the pre-computed path either they cannot be perceived by

Obstacle avoidance algorithm

while(not reached at the goal){
Follow the precomputed path while capturing data from the Kinect sensor
Continuously generate the point cloud of the current FOV
if(there is any obstacle crossing the path **and**
the distance to the obstacle is less than a threshold){
- Stop and add the current 2D space points to the world
- Rebuild the model of environment
- Compute for another path from the current pose
}
}

Table 4.2: Obstacle avoidance algorithm in Pseudo-code.

the Kinect field of view or they don't interfere with the path. The algorithm is summarized in Table 4.2.

Chapter 5

Implementation

The implementation was performed on the robot Dora [39] which is one of the CogX [32] demonstrator systems. It is a mobile robot built on a Pioneer P3-DX [40] platform equipped with a fixed laser range scanner, a pair of stereo cameras and a Kinect sensor installed on top of a pan/tilt unit (see Figure 5.1). The software architecture on the robot Dora utilized the laser range data only to build a map and handle obstacle avoidance and path planning. The problem was that the laser scanner cannot see any obstacle with any height lower or higher than its height. The idea was to enable the current obstacle avoidance to use the additional data captured by the Kinect sensor. Hence, the algorithm presented in preceding chapters has been implemented partially so that we can enable the software of Dora to effectively perform obstacle avoidance with minimum changes in the software.

5.1 Software architecture

The robot Dora utilizes the CoSy Architecture Schema Toolkit (CAST) [37] [38] which is based on Internet Communication Engine (Ice) [43] defines a framework in which different processes and threads can communicate easily and safely. The software which we worked on was labeled as *dora-yr2* indicating that the software was from the second year of the project. The architecture is divided into three major categories; *subarchitectures*, *tools* and *instantiations*. The category subarchitectures holds the main implementations in various areas such as object recognition, place classification, mapping, SLAM, obstacle avoidance and so forth whereas tools contains all the utilities required for higher level programming in the category subarchitectures. One may call tools a low level interface in comparison to subarchitectures. The last category, instantiations, includes the required configuration files for different scenarios. The core software for low level interfacing is the Player [48] which is responsible to interface between hardware and higher level software. In Table 5.1, the different subarchitectures are listed where *spatial.sa* is responsible for spatial representation of the environment and one of the components SpatialControl prepares an *occupancy grid map* as well as a *local metric map* from

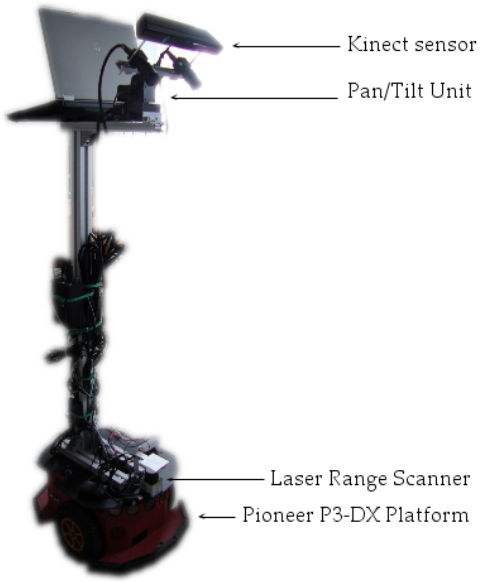


Figure 5.1: The robot Dora is one of CogX project's robots which was used for the implementation.

which obstacle avoidance is carried out. This is the component has been modified to include the data capturing by the Kinect sensor into both mentioned maps so that the robot be able to take into account the existence of obstacles which were not detectable using the laser data only.

Subarchitectures		
binder.sa	motivation.sa	vision.sa
categorical.sa	dialogue.sa	visualization.sa
coma.sa	execution.sa	planner.sa
conceptual.sa	**spatial.sa**	

Table 5.1: The content of subarchitectures where *spatial.sa* is responsible for spatial representation and obstacle avoidance is one component under this subarchitecture.

Similarly we can illustrate tools' sub-categories in Table 5.2 where the two components kinect and pointCloud have been modified such that the depth image can be returned by the corresponding point cloud server. The Kinect sensor driver and wrapper are based on OpenNI software.

Tools							
alchemy	camcalb2	castutils	d-lib	grabimg	logtools	math	rocs
beliefs	castctrl	cogxutils	genroi	**hardware**	mary	python	scripts

Table 5.2: The content of tools where *hardware* is the collection of utilities related to sensors and actuators like sonars, laser range scanner, Kinect sensor, pan/tilt unit, etc.

5.2 Method

As it is mentioned in the preceding section, the developed algorithm was integrated into Dora robot's software architecture for minimum required software change and maximum compatibility.

There are two main maps, an occupancy 2D grid map and a set of (x, y) points called local map specifically used for obstacle avoidance by another component of *spatial.sa* named as NavController. In the component SpatialControl a callback mechanism executes the function `receiveScan2d(.)` each 100 ms given the latest laser data. This data, in the original implementation, is then used to update the grid map (`m_lgm`) and local map (`m_LMap`) utilizing the function `addScan(.)` from two different classes. In our implementation, we just put the laser data in a queue and return without updating the maps for the reasons given in the following paragraphs. The component SpatialControl has a main run function named as `runComponent(.)` in which the so-called infinite while-loop is running. We implemented the task in this loop mainly because of time performance.

Because there are two different types of sensory data, 3D vision and 2D laser scan, a method to combine these effectively was required. The major problem was that the data of each sensor is captured in different time and hence with different *pose* of the robot.

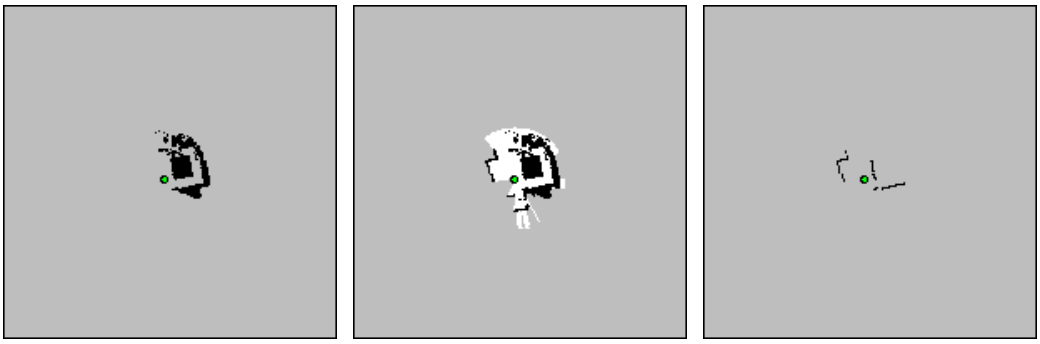
Although a SLAM process running in the component SlamProcess of *spatial.sa* provides the pose but it is obvious that the pose given in certain discrete times and thus is not providing a continuous estimation. This leads to no guaranty that the pose data at the capturing time of any kind of sensor be available directly in the sequence of the pose estimations. This issue was addressed in the software by storing the estimated pose with a time tag (which is global timing for the whole software) and make it available for other components. This approach interpolates a pose given a time and returns **true** or **false** regarding the successfulness of the interpolation. Therefore, given a depth data and laser data which are in most cases tagged with different times, sometimes there is no valid interpolation to have an estimated pose for the corresponding data. In other words, there are cases in which the data is not usable because the pose is unknown or not enough accurately could be determined.

At the beginning of the while-loop in `runComponent(.)`, a new depth image is

Combining m_lgmK and m_lgmL to produce m_lgm
$i = 0$ **while**(! end of lgm) {
$i = i + 1$
if(m_lgmK[i] is occupied){
m_lgm[i] = occupied
}
else{
m_lgm[i] = m_lgmL[i]
}
}

Table 5.3: Combining the Kinect sensor and laser data in Pseudo-code.

requested from KinectPCServer component of *tools* → *hardware*. This depth data then mapped into equivalent 2-dimensional space using the newly added function `getKinectObs(.)`. The function uses the rigid body motion of the robot through the Kinect sensor to calculate the rotation R and the translation T between the world coordinate and the Kinect sensor's coordinate. It also removes the points in the current field of view of the captured frame and then updates the corresponding grid map.



(a) m_lgmK (b) m_lgm (c) m_LMap

Figure 5.2: (a) The grid map updated by the Kinect sensor data. (b) Mixed grid map with the Kinect sensor data and laser range scanner data. (c) The result of a virtual range scanning in the grid map (b).

In the original implementation of SpatialControl there was only one occupancy grid map labeled as `m_lgm` updating with receiving each new laser data in the function `receiveScan2d(.)`. We added two more grid maps, `m_lgmL` updating by the laser data and `m_lgmK` updating by the Kinect sensor data. Then these two grid maps are combined and update `m_lgm`, see algorithm in Table 5.3 Figure 5.2

Infinite loop in `runComponent(.)`

while(true) {
  get the latest depth image data
  if(there is any laser data in queue){
  get the latest laser data
  }
  if(the pose at the captured time of laser data is available){
  update `m_lgmL` by calling `addScan(.)`
  }
  if(the pose at the captured time of depth data is available){
  update `m_lgmK` by calling `getKinectObs(.)`
  }
  combine `m_lgmL` and `m_lgmK` to produce `m_lgm`
  perform a virtual scan using `m_lgm` and update `m_LMap`
}

Table 5.4: Algorithm in Pseudo-code.

illustrates a sample situation. After updating the final grid map, `m_lgm`, we perform a virtual laser scanning using this map from where the current pose of the robot is to fill out the local map labeled as `m_LMap`. The algorithm in pseudo-code is presented in Table 5.4.

5.3 Rigid body motion of the robot

According to the discussion in section 3.1.1, a rigid body motion defines a rotation and a translation between the camera coordinate and the world coordinate. For the hardware configuration of the robot Dora, first the world point is transformed into the robot frame, then it is transformed into pan/tilt unit's frame and finally into the Kinect IR camera's frame. Figure 5.3 shows the coordinates attached to the pan/tilt unit and the Kinect IR camera.

An overall transformation yields a net rotation matrix R_{net} and a net translation vector T_{net} which describe the extrinsic parameters of the camera. Although determining the intrinsic parameters requires performing a calibration process on the IR camera of the Kinect sensor, we used the calibration data available from the team working on another Dora robot at University of Birmingham. In the next step the point cloud is generated and mapped into 2D space where grid maps are updated based on.

Figure 5.3: The Kinect sensor installed on a pan/tilt unit. For each device a frame is attached.

To show the rigid body calculation, the following rotation matrices and translation vectors with appropriate annotation are defined,

P^{w}	point coordinate in the world's frame
P^{r}	point coordinate in the robot's frame
P^{c}	point coordinate in the camera's frame
$R^{\mathrm{r}}_{\mathrm{w}}$	Euler rotation matrix of the robot w.r.t the world's frame
$R^{\mathrm{ptu}}_{\mathrm{r}}$	Euler rotation matrix of the pan/tilt unit w.r.t the robot's frame
$R^{\mathrm{c}}_{\mathrm{ptu}}$	Euler rotation matrix of the camera's frame w.r.t the pan/tilt unit's frame
$T^{\mathrm{r}}_{\mathrm{w}}$	Origin coordinate of the robot in the world's frame
$T^{\mathrm{ptu}}_{\mathrm{r}}$	Origin coordinate of the pan/tilt unit in the robot's frame
$T^{\mathrm{c}}_{\mathrm{ptu}}$	Origin coordinate of the camera in the pan/tilt unit's frame

For each defined rotation and translation we can transform the corresponding point to the target frame,

$$P^{\mathrm{r}} = R^{\mathrm{r}}_{\mathrm{w}}(P^{\mathrm{w}} - T^{\mathrm{r}}_{\mathrm{w}}) \tag{5.1}$$

$$P^{\mathrm{ptu}} = R^{\mathrm{ptu}}_{\mathrm{r}}(P^{\mathrm{r}} - T^{\mathrm{ptu}}_{\mathrm{r}}) \tag{5.2}$$

$$P^{\mathrm{c}} = R^{\mathrm{c}}_{\mathrm{ptu}}(P^{\mathrm{ptu}} - T^{\mathrm{c}}_{\mathrm{ptu}}) \tag{5.3}$$

Substitution of (5.1) into (5.2) and then into (5.3) yields the overall transformation from the world coordinate to the camera coordinate,

$$P^{\mathrm{c}} = R^{\mathrm{c}}_{\mathrm{ptu}} R^{\mathrm{ptu}}_{\mathrm{r}} R^{\mathrm{r}}_{\mathrm{w}} P^{\mathrm{w}} - R^{\mathrm{c}}_{\mathrm{ptu}} R^{\mathrm{ptu}}_{\mathrm{r}} R^{\mathrm{r}}_{\mathrm{w}} T^{\mathrm{r}}_{\mathrm{w}} - R^{\mathrm{c}}_{\mathrm{ptu}} R^{\mathrm{ptu}}_{\mathrm{r}} T^{\mathrm{ptu}}_{\mathrm{r}} - R^{\mathrm{c}}_{\mathrm{ptu}} T^{\mathrm{c}}_{\mathrm{ptu}} \tag{5.4}$$

which can be rewritten as,

$$P^{\mathrm{c}} = R_{\mathrm{net}} P^{\mathrm{w}} + T_{\mathrm{net}} \tag{5.5}$$

where,

$$R_{\mathrm{net}} = R^{\mathrm{c}}_{\mathrm{ptu}} R^{\mathrm{ptu}}_{\mathrm{r}} R^{\mathrm{r}}_{\mathrm{w}} \tag{5.6}$$

$$T_{\mathrm{net}} = -R^{\mathrm{c}}_{\mathrm{ptu}} R^{\mathrm{ptu}}_{\mathrm{r}} R^{\mathrm{r}}_{\mathrm{w}} T^{\mathrm{r}}_{\mathrm{w}} - R^{\mathrm{c}}_{\mathrm{ptu}} R^{\mathrm{ptu}}_{\mathrm{r}} T^{\mathrm{ptu}}_{\mathrm{r}} - R^{\mathrm{c}}_{\mathrm{ptu}} T^{\mathrm{c}}_{\mathrm{ptu}} \tag{5.7}$$

hence, the coordinate of the point in world's frame can be found by inverse of (5.5),

$$P^{\mathrm{w}} = R^{-1}_{\mathrm{net}} (P^{\mathrm{c}} - T_{\mathrm{net}}) \tag{5.8}$$

The measured values of the translation vectors taken from the robot's body are listed below:

$$T^{\mathrm{r}}_{\mathrm{w}} = \begin{bmatrix} x_l \\ y_l \\ 0.24 \end{bmatrix}, \; T^{\mathrm{ptu}}_{\mathrm{r}} = \begin{bmatrix} 0.156 \\ 0.0 \\ 1.1 \end{bmatrix}, \; T^{\mathrm{c}}_{\mathrm{ptu}} = \begin{bmatrix} 0.0 \\ 0.011 \\ 0.117 \end{bmatrix} \tag{5.9}$$

where x_l and y_l are the estimation pose of the robot from the localization. The corresponding Euler angles are measured / calculated based on the following definition (see Figure 5.4),

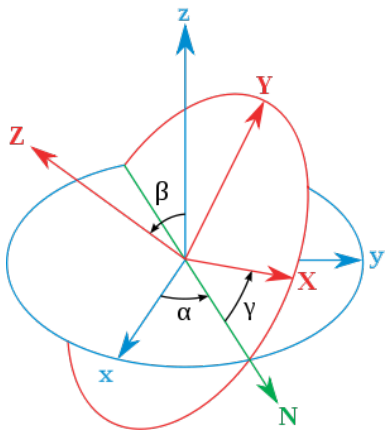


Figure 5.4: Euler angles α, β and γ. Fixed coordinate system is shown in blue (x, y, z) and its rotated system in red (X, Y, Z).

If the fixed system is denoted in lower case (x, y, z) and the rotated system is denoted in upper case letters (X, Y, Z), we define the line of nodes (N) as the intersection of the xy and the XY coordinate planes (in other words, line of nodes is the line perpendicular to both z and Z axis). Then the Euler angles are defined as:

- α is the angle between the x-axis and the line of nodes.
- β is the angle between the z-axis and the Z-axis.
- γ is the angle between the line of nodes and the X-axis.

Angles are positive when they rotate counter-clockwise and negative for clockwise rotation. With this definition, the measured / calculated Euler angles for rotation matrices are listed below:

$$R_{\mathrm{w}}^{\mathrm{r}} : \begin{cases} \alpha = 0.0 \\ \beta = 0.0 \\ \gamma = \theta_r \end{cases} \qquad R_{\mathrm{r}}^{\mathrm{ptu}} : \begin{cases} \alpha = \frac{\pi}{2} - \theta_{\mathrm{pan}} \\ \beta = -\theta_{\mathrm{tilt}} \\ \gamma = -\frac{\pi}{2} \end{cases} \qquad R_{\mathrm{ptu}}^{\mathrm{c}} : \begin{cases} \alpha = \frac{\pi}{2} \\ \beta = \frac{\pi}{2} \\ \gamma = \pi - \epsilon \end{cases}$$

where θ_r is the robot's orientation identified by the localization, θ_{pan} and θ_{tilt} are values returned by pan/tilt unit ideally. However, in practive there was not pan and although the tilt angle was set to 45 degrees but because of mechanical misalignments in the Kinect sensor, the actual angle turned out to be around 53 degrees. Another correction applied by a small angle, $\epsilon = 2.2^\circ$ subtraction from the γ to compensate slope of installation of the Kinect sensor on the pan/tilt unit.

Chapter 6

Summary and conclusion

This thesis discussed a metric approach to build a model of the environment based on the depth data provided by the Kinect sensor and constructing a collision-free local path.

We started by generating point could and mapping into 2 dimensional space. Then, a few different methods for clustering points of the 2D-map were studied and results were compared. Density-Based Spatial Clustering of Applications with Noise (DBSCAN) turned out to be able to deal with arbitrarily shaped clusters the best. Having the model of the environment, a method for path planning based on Visibility graph was developed where the robot is assumed to be holonomic and mathematically represented by a circle. The A-star search utilized to find the shortest path through the tree building by vertices of polygons as obstacles.

The algorithm integrated on the robot Dora, one of the robot demonstrators of the project CogX. Originally the obstacle avoidance algorithm relied solely on laser range scanner data which was not able to avoid colliding with obstacles standing in higher or lower heights than the laser installed. A method to combine the Kinect sensor data and laser data was developed to enable the obstacle avoidance algorithm to actually see obstacles which it was not able to see before.

6.1 Conclusion

This work revealed that although Kinect is a low-cost 3D vision device but its capability to obtain a 3D perception of the environment is invaluable. Nonetheless, it suffers from limited range detection ranging from about 50 cm to 6 meters. The accuracy of the sensor for the points further than 3.5 meters considerably decreases, though. The experience in this work also revealed that the projection pattern on a very bright or very dark area will be absorbed and thus cannot be seen by the IR camera. These limitations impose the range of applications of the Kinect sensor to be restricted to indoor environments where the level of IR absorption is not very high, and the objects closer than 50 cm are not matter of subject. This later shortcoming has a great impact on obstacle avoidance.

Continuous geometric maps can describe the environment more accurately then discrete grid based maps but the level of complexity to create and update such maps impose them computationally expensive approach. Nevertheless, to cope with unstructured environments in which maximum available free space for the robot to be able to carry out its tasks is an essential requirement, it is unavoidable to represent the environment continuously. In building such model, we faced the problem of estimating a concave-hull of a set of data points according to "the best perceived shape" concept.

The famous search algorithm A-star worked very effectively in the developed path planning algorithm. We found many papers and works in the field of path planning which indeed utilized A-star or some branch of this search algorithm. We modified standard approach of "Visibility graph" method to obtain improved performance. A safe distance to the obstacles can improve the movement of the robot as we observed in the implementation where the original path planner precomputed a path which was very close to the objects and hence made the robot struggle to follow the path without colliding with the obstacles.

We didn't find the chance to implement the algorithm completely on the robot, but even partially integration of the method into the original software architecture of the robot was challenging. CAST framework was professionally developed to deal with integration of different components, programming with different languages (C++ and Java), running in different process / threads or even remotely on the other computers effectively. Nonetheless, the level of complexity of the framework seems not to be a good choice for smaller robotic applications.

The major issue we faced in implementation was how to combine the data we obtain from the laser range scanner and the Kinect sensor according to the time difference between them and hence, in general, different poses of the robot. The idea that the localization process tags each estimated pose with the time of creation and saves them in an array was the answer to the problem. By interpolating the poses one can estimate the pose of the robot for the given time of data. This technique actually reduced the misplacement of the map and made the combined map to be useful for generating a local map which was used by obstacle avoidance algorithm. Nevertheless, the lack of a visual SLAM to compensate for the lag in the pose estimation mechanism (specifically while the robot is turning around) was apparent.

6.2 Further work

Although we tried to develop, examine and implement the ideas regarding the thesis work, but time limitation prevented us from performing all the ideas. Thus we would like to present those as further work that one may carry out in the future:

- Extending the so-called potential field obstacle avoidance method to 3-dimensional space.

- Building a 3-dimensional occupancy grid map and performing A-star search algorithm on.
- Probabilistic approach to update the map. This might enable us to use the maximum range of the sensor.
- Representation of the dense points in the 2D space with a set of rectangles (simpler polygons).
- Using the GPU power to calculate point cloud and all the transformation.
- Extending the so-called Vector Field Histogram (VFH) originally developed for sonars to 3-dimensional space using the Kinect sensor.
- Performing a visual SLAM to find out the difference in the pose between two sequentially taken frames.

Bibliography

[1] Gouko, M, *Environmental modeling and identification based on changes in sensory information*, IEEE COMPUTER SOC, 2009

[2] Gu, JJ and Cao, QX , *Path planning for mobile robot in a 2.5-dimensional grid-based map*, INDUSTRIAL ROBOT-AN INTERNATIONAL JOURNAL 38, 2011

[3] Peng Li, Xinhan Huang, Min Wang , *A novel hybrid method for mobile robot path planning in unknown dynamic environment based on hybrid DSm model grid map*, Journal of Experimental and Theoretical Artificial Intelligence, 1362-3079, Volume 23, Issue 1, 2011, Pages 5 - 22

[4] Kai Arras, Jan Perssonb, Nicola Tomatis, Roland Siegwarta, *Real-Time Obstacle Avoidance For Polygonal Robots With A Reduced Dynamic Window*, Proceedings of the 2002 IEEE, ICRA 2002

[5] Alberto J. Alvares, Gabriel F. Andriolli, et al., *A NAVIGATION AND PATH PLANNING SYSTEM FOR THE NOMAD XR4000 MOBILE ROBOT WITH REMOTE WEB MONITORING*, ABCM Symposiom Series in Mechatronics - Vol.1-pp.18-24

[6] Chong, C , Ju, HH, Chen, YZ, Cui, PY , *Path planning algorithm based on topologymap structured by scanning method*, 2005 IEEE International Conference on Robotics and Biomimetics : 117-120 2006

[7] Lavalle, S.M. *Rapidly-exploring random trees: A new tool for path planning*, Computer Science Dept., Iowa State University, Tech. Rep. TR: 98-11.

[8] Bruce, J., Veloso, M., *Real-time randomized path planning for robot navigation*, Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference, 2002

[9] Cipola R., *Handout 3: projection*, computer vision and robotics lecture note, University of Cambridge

[10] Burschka, D., Lee, S., and Hager, G., *Stereo-Based Obstacle Avoidance in Indoor Environments with Active Sensor Re-Calibration*, Proceedings of the 2002 IEEE International Conference on Robotics and Automation, Washington, DC. May 2002

[11] Dong An and Hong Wang, *VPH: A New Laser Radar Based Obstacle Avoidance Method for Intelligent Mobile Robots*, Proceedings of the 5" World Congress on Intelligent Control and Automation. June 15-19, 2004, Hangzhou, P.R. China

[12] Il-Kyun Jung, Ki-Bum Hong, Suk-Kyo Hong, Soon-Chan Hong, *Path Planning of Mobile Robot Using Neural Network*, School of Electronics Engineering, Ajou University

[13] Berg, de M., Kreveld, van M., Overmars, M., Schwarzkopf, O. *Computational geometry: algorithms and applications*, 2nd rev. edn., Springer, Berlin, Heidelberg, New York et al., 2000.

[14] Antony Galton, *Pareto-Optimality of Cognitively Preferred Polygonal Hulls for Dot Patterns*, Proceedings of the international conference on Spatial Cognition VI, ISBN: 978-3-540-87600-7

[15] Sanjiv K. Bhatia, *Adaptive K-Means Clustering*, Conference: The Florida AI Research Society Conference - FLAIRS , 2004

[16] Hepu Deng, *Artificial Intelligence and Computational Intelligence: International Conference*, AICI 2009, Shanghai, China, November 7-8, 2009, Proceedings

[17] Martin T. Wells, et al. *Data Clustering: Theory, Algorithms, and Applications*, 2007, ISBN: 0898716233

[18] Nourbakhsh, Illah R. and Siegwart, R., *Introduction to Autonomous Mobile Robots*, The MIT Press, ISBN 0-262-19502-X

[19] Murphy, Robin R., *Introduction to AI Robotics*, The MIT Press, ISBN 0-262-13383-0

[20] Choset, H. et al., *Principles of Robot Motion: Theory, Algorithms, and Implementation*, The MIT Press, ISBN 0-262-03327-5

[21] Steven M. LaValle, *Planning Algorithms*, Cambridge University Press, Available for downloading at `http://planning.cs.uiuc.edu/`

[22] Russell, S., Norvig, P., *Artificial Intelligence: A Modern Approach*, Third Edition, Prentice Hall, ISBN 0-13-207148-7

[23] Latombe, J.-C.: Robot Motion Planning, 3 Printing, Kluwer Academic Publishers, 1993.

[24] Peter M. Gruber, *Convex and Discrete Geometry*, Springer Berlin Heidelberg New York, ISBN 978-3-540-71132-2

[25] Gerard Medioni and Sing Bing Kang, *Emerging Topics in Computer Vision*, Prentice Hall., 2004, ISBN 978-0-13-101366-7

[26] Gregory E. Fasshauer, *Meshfree approximation methods with MATLAB*, World Scientific Publishing Company, 2007, ISBN 978-9812706348

[27] United States Naval Institute, *Elementary mechanics*, Naval Institute proceedings, Volume 29

[28] Steven A. Shafer, *Shadows and silhouettes in computer vision*, Springer, 1985, ISBN 9780898381672

[29] ROBERT NISBET, JOHN ELDER, GARY MINER, *HANDBOOK OF STATISTICAL ANALYSIS AND DATA MINING APPLICATIONS*, Elsevier, 2009, ISBN: 978-0-12-374765-5

[30] Daszykowski, M., DBSCAN implementation in Matlab, Department of Chemometrics, Institute of Chemistry, The University of Silesia, `http://chemometria.us.edu.pl/download/DBSCAN.M`

[31] Rare Company, `http://www.rare.co.uk/`

[32] CogX Project, `http://cogx.eu/`

[33] PrimeSense Company, `http://www.primesense.com/`

[34] Wikipedia, Kinect, `http://en.wikipedia.org/wiki/Kinect`

[35] ROS Organization, Kinect calibration, `http://www.ros.org/wiki/kinect_calibration/technical`

[36] Peter Wasmeier, hullfit, `http://www.mathworks.com/matlabcentral/fileexchange/6243-hullfit`

[37] CoSy: Cognitive Systems for Cognitive Assistants, `http://www.cs.bham.ac.uk/research/projects/cosy/presentations/cosy-overview-NAH.pdf`

[38] CAST: The CoSy Architecture Schema Toolkit, `http://www.cs.bham.ac.uk/research/projects/cosy/cast/`

[39] Dora the explorer, `http://cogx.eu/results/dora/`

[40] Pioneer 3P-DX mobile robot platform, `http://www.mobilerobots.com/researchrobots/researchrobots/pioneerp3dx.aspx`

[41] How Kinect depth sensor works stereo triangulation?, `http://mirror2image.wordpress.com/2010/11/30/how-kinect-works-stereo-triangulation/`

[42] Adafruit Industries, `http://www.adafruit.com/`

[43] Ice: Internet Communication Engine, `http://www.zeroc.com/ice.html`

[44] OpenNI organization, `http://www.openni.org/`

[45] Wikipedia, OpenNI, http://en.wikipedia.org/wiki/OpenNI Wikipedia, Pinhole Camera Model, http://en.wikipedia.org/wiki/Pinhole_camera_model

[46] OpenNI Documentation, version 1.0.0, http://www.openni.org/documentation

[47] *Path Finding Tutorial*, http://wiki.gamegardens.com/Path_Finding_Tutorial

[48] The Player Project, http://playerstage.sourceforge.net/

TRITA-CSC-E 2011:107
ISRN-KTH/CSC/E--11/107-SE
ISSN-1653-5715

www.kth.se