DEPARTMENT OF COMPUTER SCIENCE

Investigating Low-Power Computer Vision Systems Using the Microsoft Kinect to Create Autonomous Litter Bins

Jonathan Simmonds

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

May 16, 2013, CS-MEng-2013

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Jonathan Simmonds, May 16, 2013

Executive Summary

Today's blend of highly powerful, low-power single-board computers and readily available 3D-imaging cameras provides a fertile ground for the creation of advanced new robotics. This project seeks to combine the two to create a low-power computer vision system capable of replacing those used by conventional robotics applications. This is based on the hypothesis that by utilising the additional depth data provided by the Kinect it is possible to replace conventional, computationally expensive computer vision algorithms. This allows the use of single-board computers, resulting in savings in terms of power consumption.

Due to the difficulty of evaluating such claims empirically it will be investigated by means of practical experimentation, using these technologies to implement an autonomous litter bin. This bin is intended to be used in large spaces such as retail food courts to both serve as a novel and entertaining experience as well as providing litter collection services and greater convenience for customers. Designing an autonomous bin that will explore and navigate such a busy, unpredictable space is a challenging task. The project makes use of the Microsoft Kinect and the PandaBoard.

Key Achievements

- Produced an autonomous litter bin able to successfully navigate in confined, unpredictable spaces.
- Designed the bin's control system considering problems of object avoidance, path planning, target recognition and occupancy analysis.
- Analysed the power consumption characteristics of both the Kinect and PandaBoard.
- Successfully combined the two technologies to create a computer vision system.
- Investigated the strengths and weaknesses of such a system and proved it to be a valid and effective approach.
- Showed analysis of depth histograms to be a powerful technique for inferring knowledge about the 3D structure of a scene.
- Developed a marker detection and tracking system whose range exceeds that of two of the most commonly used systems.

The code for this project is publicly available at `https://github.com/jonsim/individual_project`.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Prelude

Supporting Technologies

Software

- The OpenCV library [2].
- The OpenNI library [6].

Hardware

- iRobot Create (referred to throughout as the Roomba) — http://www.irobot.com/create/.
- Microsoft Kinect — http://www.microsoft.com/en-us/kinectforwindows/.
- PandaBoard ES (referred to throughout as the PandaBoard) — http://pandaboard.org/.

Glossary

Bootstrapping
A process that continues to advance without the need of external input.

BPS
Bits per second.

Classifier
An algorithm capable of deducing the class of objects to some degree of accuracy. In the context of this project classifiers are used to decide whether a region of an image contains an object or not.

Cross-compilation
Where source code is compiled by one machine to create executable code to run on another. This is typically done when it is prohibitively slow (or even impossible) to do it on the native machine.

Dead Reckoning
The process of calculating position based on a known previous position and an approximation of movement since.

Depth Image
The 'image' returned from the Kinect's depth sensor where each pixel's value is the depth of that pixel in millimetres. While it is not technically an image, it is convenient to think of it as one.

Embedded System
A computer designed to be used as a controller in part of a larger system. Typically these are used in a real-time context.

False Negative
Incorrectly classifying an instance as negative.

False Positive
Incorrectly classifying an instance as positive.

FPS
Frames per second. The time per frame may be worked out as $\frac{1}{\text{FPS}}$.

GPU
Graphics processing unit.

Haar-like Features
Simple features which compare areas of an image based on their contrast to produce a single number output (where a higher number corresponds to better similarity with the feature). Further detailed in Section 2.2.7.

Histogram Analysis
The extraction of information by means of analysing a histogram, specifically its structure and the distribution of data within it.

Kinect
The Microsoft Kinect.

mAh
Milliamp-hour, a unit of charge capable of supplying a current of one milliamp for one hour.

Occupant Detection
In the context of this project, detecting whether or not a table is occupied.

On the Fly
An action performed during operation.

Path Planning
Refers to the calculation of a path through an environment while avoiding obstacles.

Real-Time
A constraint placed on a program requiring it to guarantee a response to a stimulus within an 'acceptable' amount of time.

Single-Board Computer
A complete computer built on a single circuit board. Typically these have a low power consumption and are used as embedded systems.

SLAM
Simultaneous localisation and mapping — a technique used to build a map of the environment while simultaneously tracking the sensors current position. Specifically refers to the computer vision algorithm used for solving the problem using a video stream.

Sliding Window Approach
A technique for searching an image by considering a single region (or 'window') which is able to be searched. This region is then moved (or 'slid') horizontally across the image to the end of the row, at which point it is moved to the next row and reset. The process continues until all possible windows in the image have been searched.

Tachometry
Measurements produced from a tachometer, a device measuring for measuring the speed at which something is rotating.

Thresholding
A simple method of image segmentation whereby all pixels below a given threshold are set to one value while those above it are set to another.

$\mathcal{N}(\mu,\ \sigma^2)$
The normal distribution with mean μ and variance σ^2.

$\mathcal{U}(a,\ b)$
The uniform distribution in the range $(a, b]$.

Acknowledgements

I must thank my supervisor, Dr. Andrew Calway for his tireless support throughout the project, generous use of his time to provide weekly meetings and for sharing his extensive knowledge of the field. Thanks also go to my parents for inspiring me to persevere and the countless hours they have spent proofreading.

Chapter 1

Introduction

1.1 The Problem

Conventional computer vision robotics applications tend to be either very simplistic in nature (with a single video camera connected to a single-board computer doing basic analysis) or very power hungry (with a laptop performing more computationally expensive analysis on the video from one or more cameras).

This project aims to achieve the best of both worlds by combining two recent advances in technology, namely the surge in availability of cheap, powerful, single-board computers (attributable to the burgeoning smart-phone market) and the introduction of off-the-shelf 3D-imaging cameras. The use of a powerful single-board computer will facilitate a cheap, easy to reproduce and low-power solution — vital in robotics applications. The addition of an off-the-shelf 3D-imaging camera will alleviate the need for many of the more computationally expensive computer-vision algorithms typically required in robotic systems, for example those involving localisation, mapping and object recognition. This is a combination which has not been widely investigated.

Single-board computers have a number of advantages over traditional computers used in robotics applications, such as laptops or remote servers. These include the fact they are cheaper to purchase (4 or 5 times cheaper), have a smaller physical footprint (5 or 6 times smaller), consume less power (up to 10 times less) and are lighter (20 or 30 times lighter); all of which are important considerations when implementing a robotic system. Off-the-shelf 3D-imaging cameras also possess a number of advantages over both traditional cameras and other depth sensing technologies. The advantage over traditional cameras stems from the inclusion of depth information, something which a number of traditional computer vision algorithms attempt to infer from the images, while advantages over other depth sensing technologies, such laser rangefinders, are principally in terms of cost (where the newer devices can be 10 times cheaper). A full analysis of available depth sensing technologies is presented in Section 2.4. By combining both of these technologies it is hoped that a system at least as capable as a traditional setup can be produced but which is smaller, lighter and cheaper.

The project will investigate the impact of these technologies experientially: by implementing an autonomous litter bin which will intelligently roam, looking for litter. This will be a much more worthwhile examination of the technology's possibilities than more conventional, empirical evaluation. This is because quantitative evaluation of the performance of a computer vision system for use in robotics is an almost futile task as all robots have their own, specialised requirements. Therefore, it is more useful to implement a full system and observe the challenges faced; allowing the performance of the computer vision system to be judged from the performance of the entire system.

The focus of the project is on implementing the autonomous litter bin; however in doing so it is hoped that some more general insight may be gained on how the combination of cheap, low-power computers and off-the-shelf 3D-imaging cameras may be exploited to create powerful computer vision systems for use in robotics. It is important that the solution is not overspecialised for the problem at hand, however, as it may make it hard to transfer the approach to a more general solution.

Producing an autonomous litter bin is not a straightforward task. There are a number of challenges involved, including: avoiding objects, localising within the environment, recognising objects and analysing this knowledge. All of this must be done to produce a system which is autonomous, robust and real-time. An additional constraint is placed on the system in terms of the processing power available from the single-board computer, which may struggle to run conventional algorithms to solve these problems. This downside is, however, countered by the additional information available from the 3D camera (in the form of depth information) which should facilitate the use of simpler solutions. The system will be implemented using a 3D-imaging camera, a single-board computer and a robotic base, the choice of which is discussed in Section 2.1.

1.2 The Challenges

As touched on in Section 1.1, the problem of producing an autonomous litter bin can be broken down into four main challenges. These are outlined below and then discussed in more detail in the following sections.

1. Object avoidance — avoiding obstacles in the environment.
2. Path planning — localising within, and calculating movement around the environment.
3. Target recognition — recognising the tables to move towards.
4. Occupancy detection — determining whether or not someone is sitting at the tables.

For this project the environment is an indoor space within which there may be inanimate objects as well as moving objects such as people. The targets within this environment will be tables which the robot will locate and move towards. The goal of the path planning is to move between targets while avoiding obstacles in as efficient and unobtrusive manner as possible. Upon arriving at a table the robot will determine its occupancy, whether or not at least one person is currently seated at it, and act accordingly.

1.2.1 Object Avoidance

This challenge focuses on avoiding both moving and stationary objects in the environment. It is possible to further subdivide this to treat moving and stationary objects separately: if moving objects could be tracked it would be possible to calculate if they were on a collision course with the robot and take appropriate evasive action. Stationary objects can be much more easily avoided as they can be detected using the depth camera and built into a map of the environment. Additional sensors may be utilised as fallback should the depth sensor fail to adequately detect an object.

This is a fundamental challenge for the project as it is vital that the bin does not drive into anything. Doing so may require human intervention and as one of the key objectives of the project is autonomous behaviour (see Section 1.3.2) this is unacceptable. Furthermore inadequate object detection or avoidance may cause damage to the robot, for example if it were to drive off a drop or into a wall. Worse still it could pose a hazard to humans if it were to drop from height on someone or knock something into someone.

This is not a new problem; work has been done on object avoidance using 3D-imaging cameras in the past (see Section 2.3 for a comprehensive list and analysis thereof) however the project's unique requirements and the computational constraint placed on it by the single-board computer mean it is a challenge that cannot be answered by an existing solution alone.

1.2.2 Path Planning

The challenge of path planning involves both building up a map of the environment (within which the robot's location is known) and calculating a route through this environment so as to best achieve

the project's goals — to make the bin available to users at tables while avoiding obstructing people's paths (see Section 1.3.1). The first part of the problem, localisation and mapping, is more traditionally solved by use of the powerful SLAM technique [7], however this has a number of downsides: namely that it is both complex and computationally expensive. The project needs neither a full map of the environment nor knowledge of its true location, only approximate knowledge of the location relative to distinctive landmarks. As such the technique is not necessary and a faster, simpler alternative can be used instead.

Path planning is a core challenge for the project as it will define the robot's ability to move through the environment with ease without either getting stuck itself or obstructing people's movement. The objectives to produce an autonomous robot with unobtrusive behaviour (see Section 1.3.2) rely heavily on the robot's calculated path not introducing it to either of these situations. It is also important that the path planning is sufficiently intelligent to match the project's requirements while not being excessively computationally expensive, something that could be a detriment to the robot's other functions, and may ultimately affect its objective to operate in real-time (see Section 1.3.2).

As with object avoidance, this is not a new problem but one that has been intensively researched (Section 2.3 goes into more detail on this), however it is necessary that a suitable solution is used for the project to match its unique requirements — specifically those with respect to performance. The addition of depth data provided by a 3D-imaging camera will also allow additional solutions to be considered.

1.2.3 Target Recognition

Target recognition refers to the challenge of recognising the targets associated with the robot's goals. In the context of this project this involves accurately recognising tables, even (especially) when there are people seated at them. This recognition is made more difficult due to the fact that it must be invariant to scale of the table (which is affected by the distance of the robot from it) and to the rotation of the table (which is affected by the position of the robot around it). Additional complexity is further added due to the camera being mounted at ground level, causing the table's perspective appearance to change with changing distance from it.

Target recognition forms another of the core challenges faced by the project, as it will allow the robot to locate its goals. Inadequate target recognition could cause it to miss a table or, worse, incorrectly identify a non-table as a table and halt next to it (causing an obstruction).

Again, target recognition is a problem which has undergone much research (Section 2.3 lists some of this) and for which there is a vast array of potential solutions, many of which are very computationally expensive. For the project's purposes it is sufficient to classify objects as either being a table or not and the depth information available to the robot should allow an efficient solution.

1.2.4 Occupancy Detection

Occupancy detection is the challenge of determining whether or not a given table is 'occupied' (i.e. there are people sitting at it). Tables identified as a result of the target recognition may be at any scale, rotation or distance. As such, the occupancy detection must be invariant to these features. It also may be necessary to re-evaluate the occupancy detection to avoid situations where occupants of the table are occluded from an initial viewpoint.

Correct identification of a table's occupancy is vital to the project as it will allow the robot to decide whether or not it is necessary to stop at any given table. Inaccurate detection may lead to the robot either failing to stop at an occupied table, or unnecessarily stopping at an unoccupied table, both of which are undesirable. Furthermore the performance of the target recognition and occupancy detection are closely linked. Robust occupancy detection will allow the robot to quickly recover from false positive results from the target recognition (where a non-table is wrongly identified as a table). Additionally, as the occupancy detection will only be performed sporadically (on objects identified by the target recognition, which will be running constantly), it is acceptable for it to be more computationally expensive than the other parts of the robot.

Occupancy detection is a problem on which little research has been undertaken. Of the research that has been done, none is in the context of this project: the research tends to consider whether or not a single given seat is occupied (occupant detection). The work that has been done has also been focused on detecting occupants from head height rather than from the floor, as this project aims to do. Further information on the related work in the field is covered in Section 2.3.

1.2.5 Human Interaction

As the finished robot will be interacting with humans in its operation it is also necessary to consider the manner of this interaction. While this is not one of the core challenges in the implementation of the system it is something which must be contemplated as it is important that the finished system is not perceived as threatening or as something which should be avoided. Furthermore it should not pose a hazard to people nearby, the speed it moves at must be carefully considered to ensure it is neither dangerous nor obstructive.

1.3 Specification

1.3.1 Aims

The final system shall:

- Roam autonomously in an indoor environment.
- Build up a map of the environment so that tables may be revisited.
- Identify tables and, at those which are occupied, come to a halt for a short period of time to allow those occupants to deposit their litter.
- Avoid revisiting tables it has recently serviced.
- Avoid obstructing walkways or people's paths.
- Prefer to act cautiously and avoid obstructive behaviour rather than taking exploratory actions which may lead to the robot reaching a table but could result in obstructions.

1.3.2 Objectives

The final system will have:

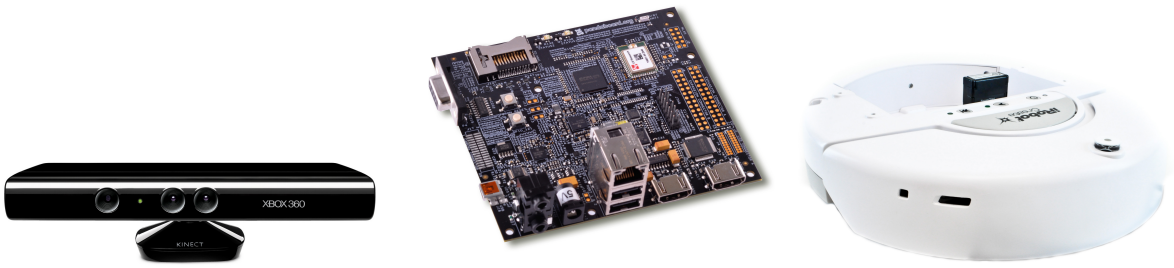
- A low-power, 3D-imaging-based computer vision solution.
- Real-time performance.
- Robust, autonomous decision making.
- Unobtrusive behaviour.
- Accurate recognition of tables.
- Robust analysis of those tables to determine whether or not they are occupied.

Chapter 2

Technical Background

2.1 The Problem

The system will comprise three core pieces of hardware: the 3D-imaging camera, the single-board computer and the motorised base. A Microsoft Kinect has been chosen as the 3D-imaging camera (described in Section 2.2.1), a PandaBoard as the single-board computer (described in Section 2.2.2) and an iRobot Create as the motorised base (described in Section 2.2.3). The existing algorithms which have been used by the project are detailed in Sections 2.2.4 on. The proposed assembly of the robot from these components is shown in Figure 2.2.



(a) A Kinect. (b) A PandaBoard. (c) An iRobot Create.

Figure 2.1: A collection of images of the three main pieces of hardware that are used on the project (described in Sections 2.2.1, 2.2.2 and 2.2.3 respectively). The images are taken from promotional material available for each of the devices.

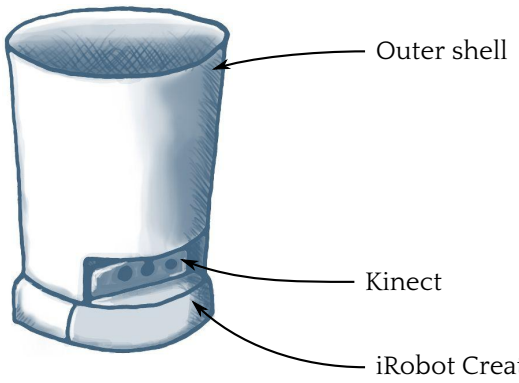


Figure 2.2: A rough sketch showing the proposed structure of the robot with the various components labelled. The PandaBoard and other electronics will be secured behind the Kinect.

The Kinect has been chosen for many reasons: because it is widely available and used so is well documented; because it is cheap; because it is low-power (an analysis of which is presented in Section 8.1.3) and because it is portable. Further justification and analysis of the alternative technologies available are described in Section 2.4. The PandaBoard has been chosen for a variety of similar reasons (discussed in more detail in Section 2.2.2), namely that it is cheap, low-power and computationally powerful compared to other similar products. The iRobot Create has been chosen because it is, again, a commonly used device for the prototyping of robotic systems and because it provides an accessible interface through which it may be controlled (further described in Section 2.2.3).

As outlined in Section 1.2, the system will be split into four main modules:

1. Object avoidance.
2. Path planning.
3. Target recognition.
4. Occupancy detection.

Each of these modules will solve a specific challenge faced by the project. The interaction of these modules is outlined in the system flowchart presented in Section 3.1 (Figure 3.1).

2.2 Technical Basis

2.2.1 Kinect

The Kinect is a 3D-imaging camera (also known as an RGB-D camera) originally released in 2010 as a game controller for the Xbox 360 entertainment system. It works using a combination of techniques, described below, to calculate depth and has three lenses (shown in Figure 2.3) which house an infrared (IR) projector, an infrared camera and a standard (RGB) camera. The RGB camera is not used for depth computation.

The Kinect primarily works using an adaption of a technique known as structured light scanning. The conventional approach to this technique works, in principle, due to the fact that projecting a pattern onto a 3D shape causes the pattern to become distorted from viewpoints other than the projector. In more depth, the structured light technique traditionally works by projecting a regular, known pattern (hence structured) onto a 3D surface and capturing an image of the result from a different viewpoint. Then, given the distance and rotation between the two viewpoints is known (labelled in Figure 2.4 as the triangulation base), the observed pattern can be used to reconstruct the 3D surface. Figure 2.4 illustrates this concept where a pattern is projected onto an object and the camera observes the result (with a single strip highlighted to exemplify the distortion observed). It should be noted that both the distance and rotation between camera and projector are greatly exaggerated for illustrative purposes; in the case of the Kinect the distance between the camera and projector is 75 mm (which can be seen on Figure 2.3) with no rotation.

In practice the Kinect uses an alteration of this technique: rather than projecting a regular, visible pattern it projects a static, pseudo-random pattern of dots in infrared. An example of the pattern of dots used by the Kinect is shown in Figure 2.5. Individual dots can then be identified by observing the pattern of dots nearby, which, due to the pseudo-random nature of the pattern, is unique. The deformation of the pattern can then be worked out as in the traditional structured light technique by considering the shift each individual dot has gone through compared to the original projection.

The accuracy of the Kinect's depth calculation is improved using a modification of a technique called 'depth from defocus' which works on the principle that the more blurred an object is in an image the further away it is [8]. The Kinect exploits this property by using an astigmatic lens in front of the IR projector [9]. An astigmatic lens has different focal lengths on two different perpendicular planes, causing the horizontal and vertical components of a projected image to be in focus at different distances. In the case of the Kinect this causes circular dots to be projected as ellipses whose orientation depends

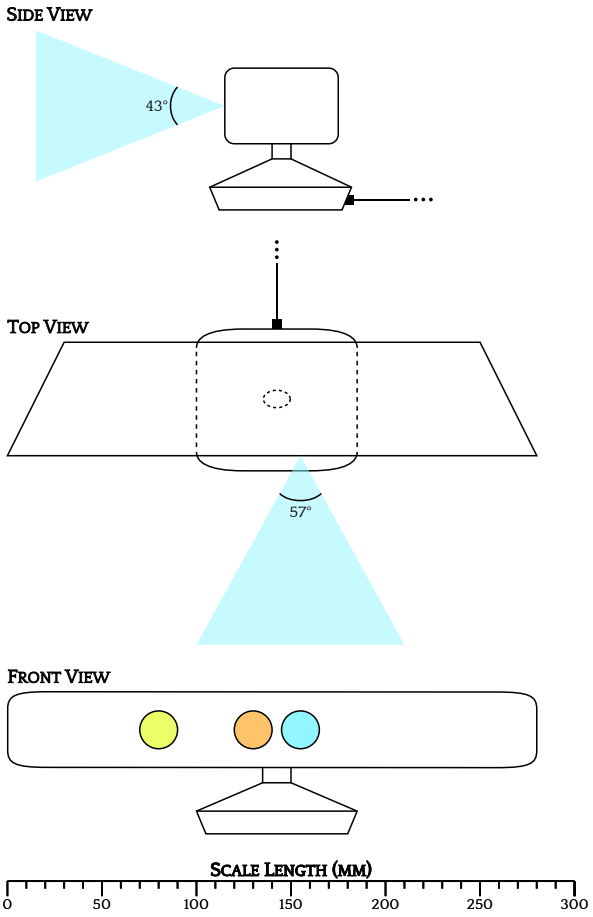


Figure 2.3: A set of 1:4 scale drawings of the Kinect. Starting from the left, the lenses shown on the front view correspond to: an IR projector (green), a standard RGB video camera (orange), and an IR camera (blue).

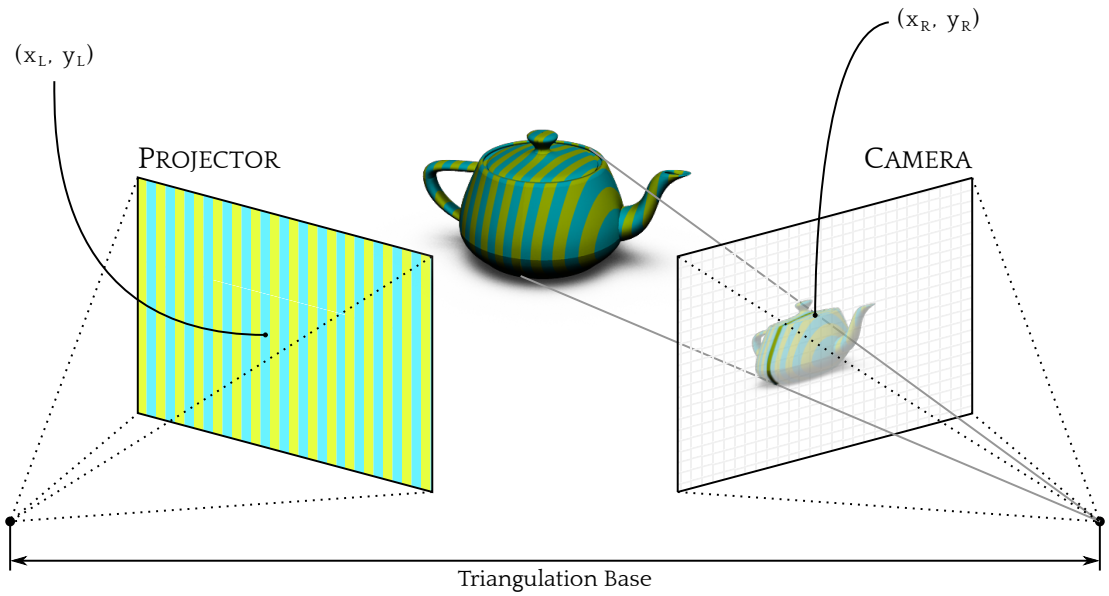


Figure 2.4: A diagram illustrating use of the structured light technique to infer information about the shape of a 3D surface.

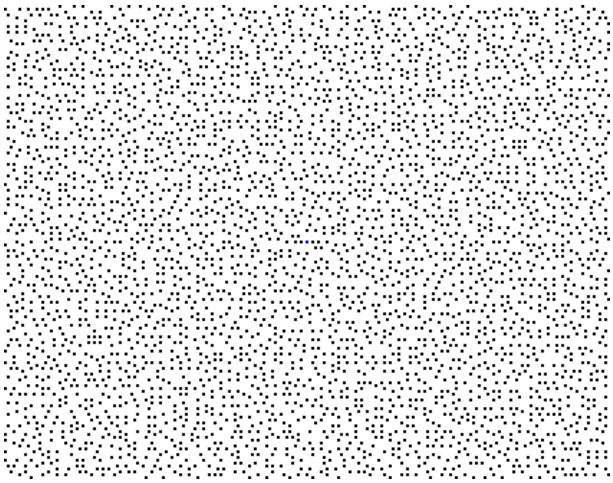


Figure 2.5: A representation of $1/9^{th}$ of the Kinect's projected, pseudo-random pattern. The full pattern is generated by replicating this pattern in a 3×3 grid at different brightnesses. This image is a printable version of one generated by *azt.tm* [1].

on the distance from the lens, as shown in Figure 2.6. The exact shape of the projected dot may be observed and used to additionally improve the Kinect's accuracy.

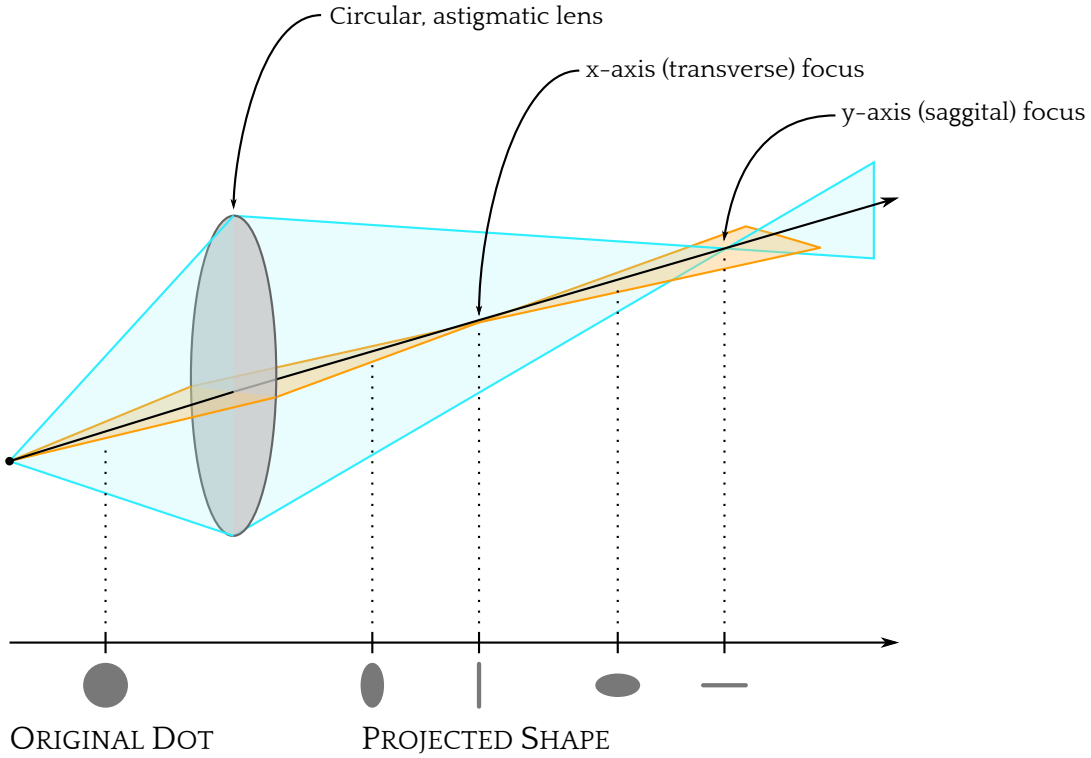


Figure 2.6: A diagram showing the effect an astigmatic lens has on propagating rays. The shape of a circular dot projected by this lens at different distances along the light's path is also shown.

The combination of techniques used by the Kinect allow it to measure depth with an accuracy of ±4 cm at a depth of 5 m, decreasing quadratically with depth [10]. All depth computations are able to be performed in real-time, at 30 frames per second and with a 640×480 pixel resolution using the on-board chip within the Kinect. Analysis of the Kinect's power consumption is presented in Section 8.1.3.

Despite the powerful and accurate technology used in the Kinect it is far from perfect. One of its main drawbacks is the short operating range. The Kinect can detect objects from 0.4 m to 10 m away (re-

strictions imposed by the drivers used), however it is only accurate for objects between 0.5 m and 5 m (as verified and used for testing by Khoshelham and Elberink [10]). This range is illustrated in Figure 2.7.

A further problem is the noise present in the data. A value of zero depth may be returned if the pixel is: in shadow; too close to the sensor; too far from the sensor or unable to produce a sample. The pixel may be in shadow if there is an object in the foreground, causing the objects behind it to be in shadow to the IR projector. This shadow is then visible from the IR camera's perspective as it is in a different position. A pixel will be unable to produce a sample when the projected pattern becomes lost, something which can happen either as random noise (which will change from frame to frame) or as a result of a particular surface material and viewing angle. As a quick comparison, a single frame has 307,200 pixels and, in the context of this project (with the Kinect mounted at floor height), typically 65,000 of those have a value of zero (values averaged from the 10 frames visible in Figure 4.3 and rounded to the nearest 1000). This is equivalent to 21% of the total pixels; obviously a considerable limitation. It is worth noting, however, that all noise has a value of zero and so can be easily identified and removed. Non-zero values can therefore be relied upon (after taking into account the measurement accuracy), unlike in colour images from cameras where the noise is similar in colour to that of the actual object.

Furthermore the Kinect's readings become significantly more noisy in an outdoor environment due to additional IR radiation from the sun, although this depends heavily on the weather conditions and time of day. El-Laithy *et al.* [11] showed that during the afternoon the Kinect was unable to produce any readings, however in the evening in just a 15 minute window the result went from being unusable to usable. While this is not a concern for this project, it is a limitation for other robotics applications.

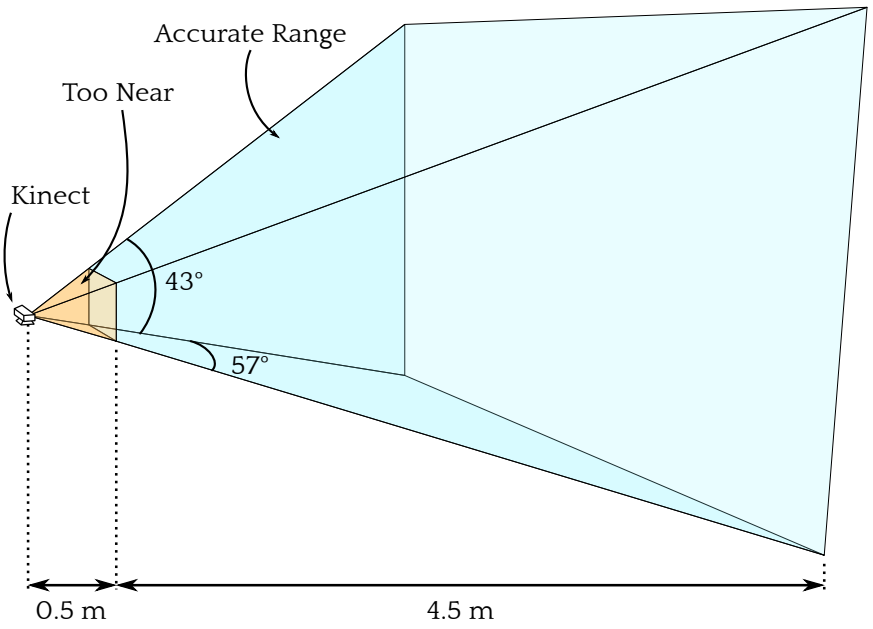


Figure 2.7: A scale diagram showing the range for which the Kinect is accurate. It can return values in the range 0.4–10 m, but is only accurate for 0.5–5 m.

The depth information produced is a 640×480 resolution collection of pixels, where the value of each pixel represents its depth in millimetres. This project visualises the depth field using a 640×480 image where each pixel is given a colour corresponding to its depth. Colours were used to represent the various depths rather than a grayscale mapping as it allows a far greater range of numbers to be displayed, making subtle details and small variations more apparent. An example of one such depth image, along with the scale against which it is mapped, is shown in Figure 2.8.

Other Kinect-like devices are available on the market, specifically the ASUS Xtion series. These have been developed in conjunction with PrimeSense and use the same PrimeSense technology as the Microsoft Kinect [12]. The key differences between the Kinect and the Xtion series are that the ASUS devices are smaller and do not have a motorised base, allowing them to be powered uniquely over USB, whereas the Kinect must be powered using a mains power supply (an analysis of the Kinect's power

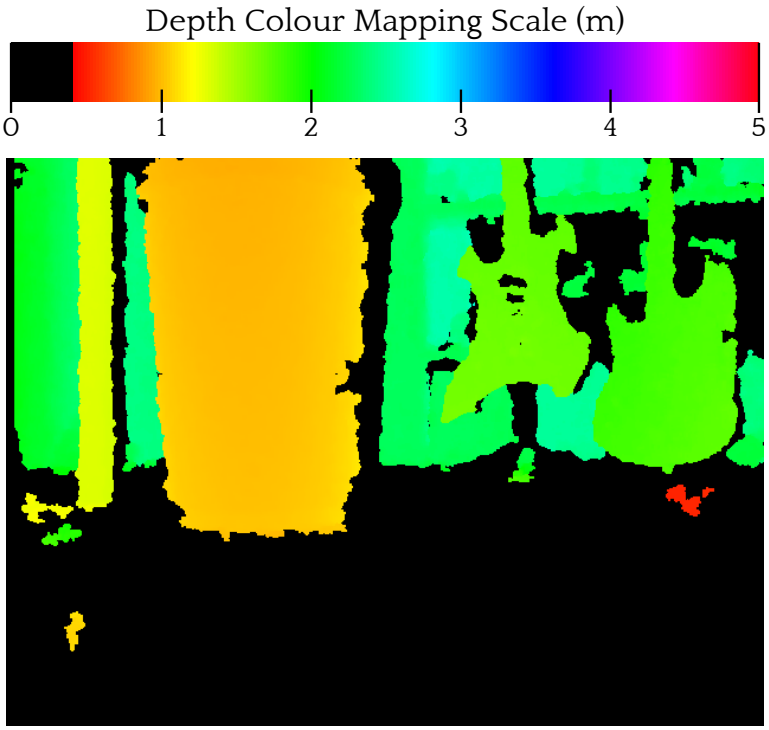


Figure 2.8: A typical depth image from the Kinect.

consumption is given in Section 8.1.3). As this project does not make use of the Kinect's motorised base the ASUS devices could be used interchangeably.

2.2.2 PandaBoard

The PandaBoard is a low-power single-board computer. The cost of this is partially subsidised by Texas Instruments (TI), a sponsor of the project and a manufacturer of some of the board's components, including the main processor. This subsidy is designed to encourage the board to be adopted as an open source development platform [13]. The PandaBoard ES, the device actually used for this project, is a newer version of the PandaBoard using slightly upgraded components and is referred to throughout this document simply as the PandaBoard.

Various potential single-board computers were considered for this project, a comparison of which is shown in Table 2.1.

Board	Size (mm)	Chip	RAM (MB)	Power (W)	Price (£)
RaspberryPi (rev 2)	86 × 54	700 MHz ARM11	512	3.5	27.21
BeagleBone	85 × 53	720 MHz Cortex-A8	256	1.5	58.65
BeagleBoard (rev C5)	76 × 76	720 MHz Cortex-A8	256	2.0	112.33
BeagleBoard XM	83 × 83	1.0 GHz Cortex-A8	512	3.7	125.68
PandaBoard	114 × 102	1.0 GHz Cortex-A9 (dual-core)	1024	5.0	114.07
PandaBoardES	114 × 102	1.2 GHz Cortex-A9 (dual-core)	1024	5.0	119.31
ASUS AT5IONT-I [1]	170 × 170	1.8 GHz Intel Atom (dual-core)	4096	24.0	137.53

Table 2.1: A comparison of different single-board computers.

The PandaBoard (ES) was ultimately chosen due to its superior processing power and memory at the small cost of both additional power consumption (a full analysis of which is presented in Section 8.1.3)

[1] Technically not a single-board computer but a more conventional, ITX-factor motherboard — shown both for comparative purposes and as a more powerful alternative.

and additional monetary cost. It also boasts a number of useful additional features over other boards, such as a wireless networking chip which allows the robot to be controlled and debugged remotely.

The PandaBoard was setup using a 32 GB memory card with Ubuntu 12.04 installed. It was configured to run headless (without a monitor attached) and automatically connect to a laptop using the WiFi connection when turned on. The laptop was automatically located over the network and once connected both machines saved each others IP addresses. These were then used by the program to configure connections for streaming depth and/or colour video (compressed to JPEG frames for faster transmission). The PandaBoard has 2 USB connections, which were used to connect to the Kinect and iRobot Create, as well as an RS232 serial connection which was not used. Development was done on the laptop and transferred over to the PandaBoard for compilation. It was decided not to use cross-compiling due to the complex arrangement of libraries (specifically OpenCV [2] and OpenNI [6]), both of which had to be compiled from source and customised to run on the system. The PandaBoard was connected to a 10,000 mAh battery, giving it an operating time well in excess of the iRobot Create's. An analysis of the power consumption of the PandaBoard is provided in Section 8.1.3.

2.2.3 iRobot Create

The iRobot Create is a robot manufactured by iRobot based on their Roomba product: a robotic vacuum cleaner. The Create is very similar to the Roomba, however lacks the vacuum components and has a rear cargo bay instead. The Create is explicitly designed for use as a robotics platform and has a serial port through which commands may be issued to it as well as an exposed and well documented interface for these commands. Due to the similarities with the Roomba product the Create is frequently and interchangeably referred to as a Roomba, something which is done throughout this document.

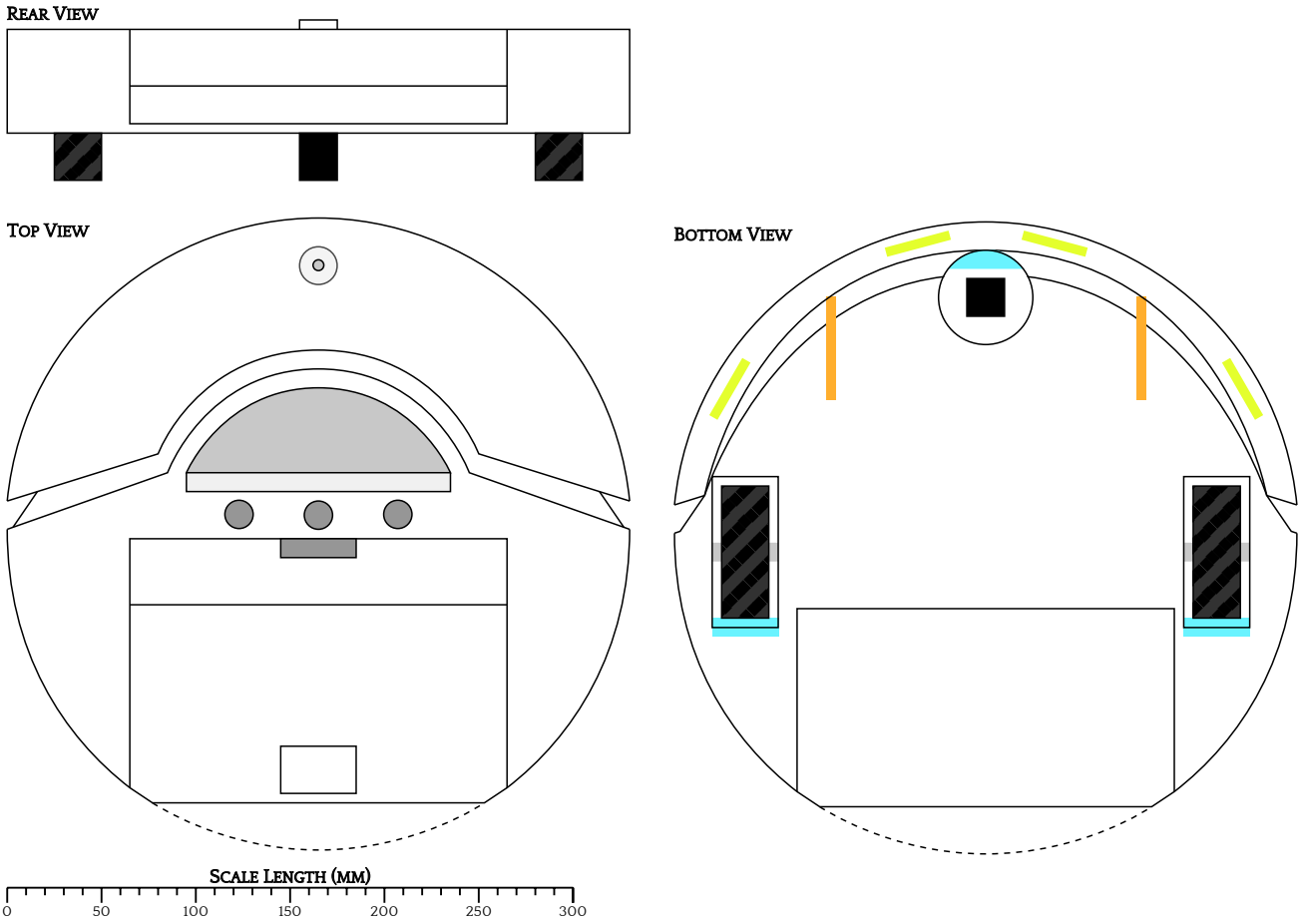
The Roomba is a circular robot with two driven wheels on either side and a caster wheel at the front. This allows it to turn within its own footprint, an important consideration when the robot only has line of sight in front of it. Furthermore this round, low-profile shape is ideal from the point of view of constructing an autonomous litter bin, allowing the entire device to fit inside the footprint of a bin. It is equipped with a number of sensors (shown in Figure 2.9), including:

- A sprung front bumper with sensors on the left and right which can be activated independently. This has approximately 14 mm of travel with the sensors activating after about 6 mm.
- Four infrared cliff sensors mounted along the front bumper which detect sudden drops in ground ahead of the robot.
- Wheel drop sensors on each of the three wheels. The wheels are mounted on sprung arms underneath which there is a small switch which detects when this arm extends, causing the wheel to 'drop' below its normal position (i.e. if the weight applied on it is lessened). This would occur if the wheel were driven over a drop.

In addition to these sensors, which are vital for avoiding obstacles, the Roomba can record how far each of the wheel motors has travelled — something which may be calculated by knowing both the rotation undergone by the motor and the wheel's radius. This allows the Roomba to slowly map the environment using dead reckoning (described in Section 2.2.6), providing the error of these measurements is not too great.

The Roomba also has other sensors for battery monitoring, detecting walls and detecting the charging station (which has two infrared 'beacons' to help the Roomba line up with it).

Finally the Roomba has a serial port on it which allows commands to be issued to it using the "iRobot Create Open Interface Protocol" [14]. This is a basic binary protocol where each command is represented by a single byte code followed by any applicable parameter bytes. If the command has a response this is returned as soon as possible, during which time no further commands may be sent. Using this protocol a host may, among other things, retrieve the value of any of the Roomba's sensors (as described above) or issue commands to drive either (or both) of its wheels. The Roomba communicates by default at 57,600 bits per second (BPS) which, assuming each command takes 10ms to be responded to and features a single command byte and two response bytes (typical of sensor reading commands which will make up the bulk of communication), allows for a theoretical maximum throughput of 96 commands per second.



(a) Top and Rear Views.

(b) Bottom View.

Figure 2.9: A set of 1:4 scale drawings of the Roomba. The coloured markings on the bottom view correspond to: cliff sensors (green), wheel drop sensors (blue) and bumper sensors (orange).

The Roomba's battery power may be easily accessed via a provided connector. The Kinect may be powered by this as it outputs 12 V at 1.5 A when the Roomba is powered on. An analysis of the Kinect's power consumption is provided in Section 8.1.3.

A shell (shown in Figure 2.2) was built for the Roomba using a 50 L dustbin, from which the bottom was removed and a hole cut in the wall for the Kinect to be mounted in (taking into account the field of view, as visualised in Figure 2.7). A false bottom was then fitted above this, to which brackets were attached so it could be connected to the Roomba. The shell can be seen in Figure 8.1.

2.2.4 Point Clouds

A point cloud is the set of 3D coordinates generated by 3D-imaging cameras. The output of the Kinect's depth camera is a 640×480 image where each pixel is a number in the range 0–10,000 representing the depth of that pixel from the camera plane in millimetres. This can be visualised as shown in Figure 2.10.

The Kinect's depth image provides the value of Z_P, and the pixel's position gives values for x_P and y_P. Thus the full 3D coordinate may be calculated using similar triangles as follows:

$$X_P = \frac{x_P \cdot s_x \cdot Z_P}{f} \tag{2.1}$$

$$Y_P = \frac{y_P \cdot s_y \cdot Z_P}{f} \tag{2.2}$$

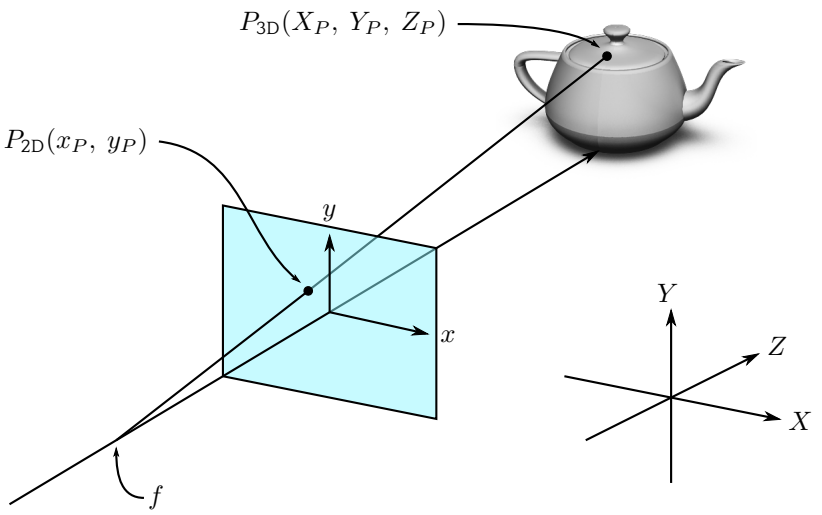


Figure 2.10: A diagram showing the projection of a single point from the 3D space onto a 2D image plane using a camera with focal point f.

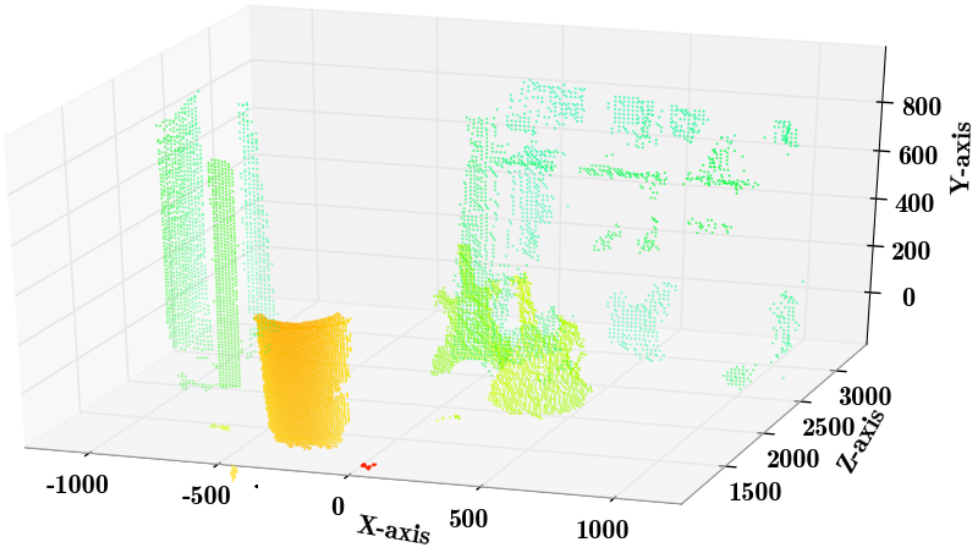


Figure 2.11: A point cloud generated from the image shown in Figure 2.8.

where s_x and s_y are the respective horizontal and vertical sizes of a pixel at the camera plane. In the Kinect $s_x = s_y = 0.2084$ mm, and $f = 120$ mm. Calculating the 3D coordinates of all the points then gives an output similar to that shown in Figure 2.11. This is a powerful technique, utilising the Kinect's data to allow the real world position of objects to be calculated from their image coordinates.

2.2.5 Circular Buffers

One of the key data structures used in the project is the circular buffer. This is a buffer with a fixed size which does not require elements to be moved as it is updated, making it ideally suited to handling data streams. As it acts as a continuous stretch of memory with no fixed start or end it is easily visualised as if both ends were connected (as shown in Figure 2.12); however it is actually implemented on a fixed-length block of memory which keeps track of the head and tail.

In Figure 2.12 new frames would be inserted at the 'Head' position causing the 'Head' to be moved round one space, potentially also moving the 'Tail' round one space if the buffer is full. In this case the oldest

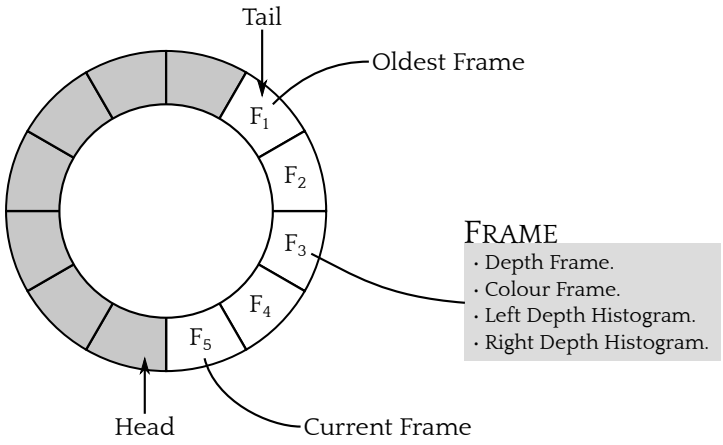


Figure 2.12: A diagram conceptually showing a circular buffer designed for storing frames.

element becomes lost.

The advantage of a circular buffer is that, because it is used as a continuous ring of data, new items are simply inserted at the head, regardless of where that actually falls on the underlying memory structure. This means that the structure takes $O(n)$ space, which, importantly, may be allocated as a contiguous chunk of memory and never reallocated — allowing for a fast implementation in practice. Additionally insertion, deletion and retrieval operations can all be performed in $O(1)$. Disadvantages of a circular buffer include: that it is a fixed size, and so cannot be grown or shrunk; that items cannot be deleted except from the end of the buffer and that, if the buffer is full, old items are overwritten as new ones are added. All of these restrictions are justifiable for use as a data buffer.

This project makes use of a circular buffer to store the frames and their meta-data as they are captured. The frame buffer is one of the core components in the implementation of the robot and its fast performance underpins that of many other components. The frame buffer's size is known in advance and is dictated by the operations which need to be performed on it, for example the histogram analysis described in Section 4.2.

2.2.6 Dead Reckoning

Dead reckoning is an intuitive method of calculating position based on a known previous position and an estimate of distance travelled and angle of travel. This produces an estimate of the current position which can be subsequently used as a starting position for dead reckoning on further movements. This is a cheap and easily performed operation requiring very little knowledge, however as it involves using estimates to make further estimates it is subject to cumulative errors, making it unsuitable to be solely relied upon. In the case of this project the distance travelled can be determined from the circumference of the wheels and the rotation they have undergone. Likewise angle turned can be calculated by knowing the distance between the wheels and the distance each has travelled. This allows an estimate of current position to be calculated as follows:

$$\theta_{\text{current}} = \theta_{\text{previous}} + \Delta\theta \tag{2.3}$$

$$x_{\text{current}} = x_{\text{previous}} + d \sin \theta_{\text{current}} \tag{2.4}$$

$$y_{\text{current}} = y_{\text{previous}} + d \cos \theta_{\text{current}} \tag{2.5}$$

where $\Delta\theta$ is the angle rotated through (where positive values indicate clockwise rotation and negative values indicate anti-clockwise rotation) and d is the distance travelled. This assumes the rotation is applied before the translation, an ambiguity which can be avoided by performing either a rotation *or* a translation, but never both at once.

To avoid the errors accumulated by dead reckoning it is necessary to re-localise every so often. This requires either the use of known landmarks or other sensor data to test the expected readings in the

estimated position against the actual readings. The frequency with which the re-localisation needs to occur depends on the accuracy of the measurements used to carry out dead reckoning as well as the accuracy required. This project makes use of tables, which are identified using the camera, to adjust position and 'reset' any errors incurred.

2.2.7 Viola-Jones Object Detector

The Viola-Jones object detection framework [15] is a real-time approach for classifying objects in images using machine learning. It works by making use of so called 'Haar-like features': very simple features made up of just two or three binary rectangles. Examples of such features are shown in Figure 2.13. They are evaluated by calculating the sum of the pixels in the 'white' region and subtracting the sum of the pixels in the 'black' region, giving a single number result. Importantly the sums of rectangular features may be calculated in constant time by use of an integral image. In this each pixel's value is given by the sum of all pixel values in the original image above and to the left of it, allowing the area of any rectangle to be calculated from the values of the four corner points. Rotated rectangles may also have their area calculated in this way, however an additional integral image is required, itself rotated by the same amount. This project makes use of the OpenCV [2] implementation of the Viola-Jones detector, the complete set of Haar-like features used for which is shown in Figure 2.13.

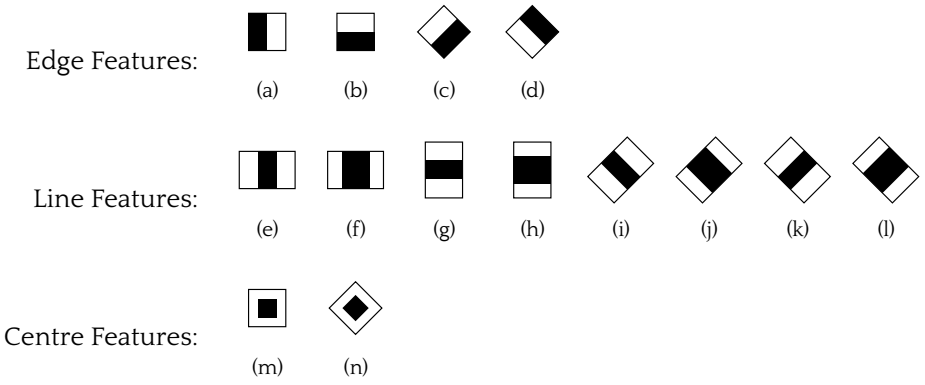


Figure 2.13: The complete set of Haar-like features used by the OpenCV [2] implementation of the Viola-Jones object detection framework. This project only makes use of a subset of these features, specifically (a), (e) and (f), for reasons explained in Section 6.2.

These simple prototype features can be scaled to cover different areas of the image and as such represent a considerable number of actual features: a 24×24 window can produce in excess of 110,000 actual features [15, 16, 17]. Having defined this set of features, Viola and Jones then used a machine learning approach to determine the features that were useful for identification of the object in question. So that the classifier may determine which features are important a large number of example images must be presented, including both positive (those which contain the object in question) and negative (those which do not) samples. The training set size is typically in the order of thousands: Viola and Jones used approximately 5000 positive images and 10,000 negative images, while similar numbers have been used by other work in the field (Lienhart *et al.* [16] used 5000 positive and 3000 negative).

The advantages of using Haar-like features are their simplicity and the speed with which they can be evaluated, however they are let down by their crude and imprecise nature. These properties can be well exploited by a boosted classifier. Classifier boosting is a machine learning technique whereby a single, 'strong' classifier is built up of a large number of simple, 'weak' classifiers which do not have a great accuracy on their own but serve to do better than guessing. The implementation used makes use of the AdaBoost (or adaptive boosting) technique which takes a large number of examples, both positive (those which contain the object in question and for which the classifier is expected to register a detection) and negative examples — examples. AdaBoost is a good choice because it 'adapts' the weights given to the training set so that subsequent classifiers in the chain focus on correctly classifying examples which previous classifiers got wrong. Each classifier is then made up of a single Haar-like feature and a threshold. More formally, the classification of input image x by the i^{th} weak classifier

$C_i(x)$ can be given in terms of a single feature $f_i(x)$ and a threshold τ_i as follows:

$$C_i(x) = \begin{cases} 1 & \text{if } f_i(x) < \tau_i \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

These classifiers are not designed to be particularly accurate on their own: each one is trained to eliminate at least 50% of non-object patterns (true negatives) and no more than 0.5% of object patterns (false negatives). It is important to note that any given classifier may be adjusted to have a lower false negative rate by raising the value of τ_i (until the extreme where τ_i is higher than all possible values and all instances are classified positively). This will have the effect of additionally increasing the false positive rate where the extent of this increase defines the effectiveness of the classifier and gives a metric by which to compare it with others. The theoretical false positive rate (also called the 'false alarm rate') and true positive rate (also called the 'hit rate') can be given by:

$$\text{False Alarm Rate} = 0.5^{\text{number of stages}} \tag{2.7}$$

$$\text{Hit Rate} = 0.995^{\text{number of stages}} \tag{2.8}$$

For the purposes of this project 15 stages were used, giving a theoretical false alarm rate of $0.5^{15} = 3.1 \times 10^{-5}$ and a hit rate of $0.995^{15} = 0.928$.

Each stage of boosting requires the generation of a number of classifiers and subsequent evaluation of these against the training set (using a constant threshold value). The training set is weighted such that instances incorrectly classified by previous stages are more important to get correct. The best performing classifier from the set is then selected and a threshold value selected that minimises false negatives. If this fails to meet the minimum accuracy laid out above (rejecting at least 50% of non-object instances and at most 0.5% of object instances) additional classifiers are generated using this best candidate and an additional random feature. This process of adding features to generate increasingly complex classifiers is repeated until the minimum accuracy is achieved, at which point the final classifier is added to the stage and training on the next stage is begun. With each stage it becomes more difficult to split the training set due to the re-weighting favouring the harder examples; as a result requiring the generation of a greater number of classifiers and a longer training time to reach the minimum accuracy. The complexity of the classifiers may be further increased by not just adding features independently but also using more sophisticated structures to store them, thus learning relationships between features. This allows classifiers to use different features to analyse different inputs, reducing the total number of features evaluated but allowing the features to be more specialised. The effect of using such structures to store features is examined in Section 6.3.

Once trained, classifiers are deployed using a sliding window approach where only a small region, or window, of the image is considered and objects are searched for within it. This window is scanned horizontally across the image, moving down with each completed row. Once a full pass of the image has been made, the window is scaled up and the process repeated. This allows for scale-invariant detection of objects, a necessity when images may be any size and objects may be any distance from the camera. Scaling up stops when the window exceeds the size of the image in any dimension. Due to the considerable overlap between windows, false positives may be further reduced by requiring a minimum number of neighbouring windows to detect the object to register a true detection. The effects of both this and the scaling rate are investigated in Section 6.3. In the tests carried out by Viola and Jones on 384×288 pixel images an average of 577,552 windows were considered for each image.

The cascade of classifiers produced has the considerable advantage that failure at any stage allows rejection of the window (illustrated in Figure 2.14). This engenders very efficient behaviour, allowing rejection of most windows very quickly — Viola and Jones' complete cascade had 38 stages with a total of over 6000 features but performed an average of just 10 feature evaluations per window. This is important because out of the 577,552 windows considered per image the overwhelming majority will not have contained an object instance.

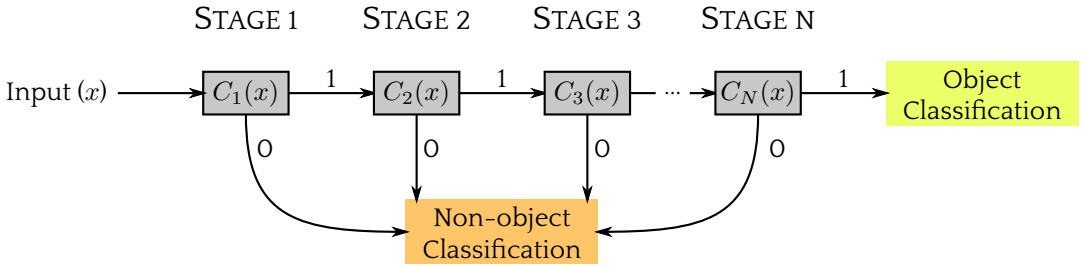


Figure 2.14: A diagram illustrating the cascade of classifiers employed by the Viola-Jones object detector.

2.3 Related Work

2.3.1 Inspiration

One of the key inspirations for this project is the 'DustCart' robot, part of a wider $3.9m EU research program called 'DustBot' which aimed to use robotics to improve urban hygiene [18]. The 'DustCart' robot was trialled for two months in the Italian village of Peccioli which has roads too steep and narrow for conventional refuse collectors. It served to gather and dispose of refuse, monitor air quality and provide civic information (e.g. bus timetables) to passing people. Residents could summon the robot via telephone causing it to come and collect rubbish from their doorstep and return it to a drop-off point. The robot worked using a hard-coded map of the town within which it was able to localise itself from a number of beacons distributed around the map (35 were spread on the 300 m route). Additionally WiFi points and cameras were installed to allow the robot to remain connected to, and be monitored from, the control station, from which it could also be remotely controlled if necessary (which it was on 37% of services). It made use of collision avoidance systems to prevent it running into objects or people, and a special lane was added to the roads to prevent interference with conventional traffic.

2.3.2 The Kinect in Robotics

Research has been carried out to investigate the use of the Kinect as a computer vision solution in real-time robotic systems. One of the first examples of this was by Stowers *et al.* [19] who mounted the Kinect facing downward and used it for altitude control in a quadrotor — an unmanned aerial vehicle (UAV) which typically consists of a cross-frame with four rotors mounted on projecting struts. While this was a real-time application the only on-board processing was a microprocessor to handle motor control, the Kinect's processing was done by a laptop connected via USB, severely limiting the device's flight range. The Kinect was found suitable for this purpose, although it was discovered the accuracy was dependent on the material of the floor beneath the quadrotor.

The use of the Kinect for robotic navigation in tightly packed and highly dynamic environments has been examined by Draelos [20], who made use of additional lenses to shorten the Kinect's operating range (a limitation discussed in Section 2.2.1 and illustrated in Figure 2.7). The Kinect was then re-calibrated to ensure accurate measurements could be maintained at shorter distances. The aim of using such optics was to allow the robot to get closer to, and handle interaction with, close-range obstacles. The project was partially successful: following the lenses addition and re-calibration accurate measurements were obtained at distances down to 0.3 m, with an improvement in close-range depth resolution of 30%. This was, however, achieved at the expense of long-range depth resolution which was reduced by an equivalent 30%. The lenses also severely impacted the Kinect's field of view. The robotic application the system was tested with successfully navigated a maze both with and without the modified optics; although time taken was increased on all tests with the additional lenses. The system was implemented using a combination of a netbook (to control the robotic base and extract data from the Kinect) and a desktop computer (to process the data) linked using an Ethernet cable.

Very little academic use of the Kinect in conjunction with a low-power, single-board system has been

found. One of the few examples of this is by Kepski and Kwolek [21] who used a PandaBoard and Kinect in conjunction with a wearable accelerometer and gyroscope to implement a fall detection system for the elderly. The Kinect, which was in a fixed position, was used to reduce the frequency of false positive detections associated with noisy or ambiguous sensor readings — for example those generated when sitting down quickly. The combination allowed a 100% accuracy rate in a home environment.

Outside of academic literature there are equally few examples of the Kinect used with a single-board system. One of the first and best documented uses found was the TiroKart project [22] which has unfortunately been taken offline over the course of this project, although an overview is still available [23]. This experimented using the Kinect and PandaBoard to automate an off-road racing go-kart. The PandaBoard served as an interface to the sensors and go-kart controls as well as providing low-level functionality to the system. The main processing was then done using a "remote PC cluster". The SyntroKinect project [24], run in part by the same author, has also attempted the combination although while their method is well published, their results are not. This project is designed to be a vision module for a larger robotics project, again appearing to use a single-board computer as a low-level controller interfaced by a more powerful computer performing high-level data analysis.

2.3.3 Object Avoidance

The Kinect has also been used in object avoidance systems before, for example by Mojtahedzadeh [25] who examined the usage of the Kinect in a collision avoidance system for the "Dora" robot, part of the CogX project [26]. This approach involved building a point cloud from the Kinect's depth image (a technique described in Section 2.2.4) which was then clustered into discrete groups assumed to be obstacles. Finally these obstacles were used to augment data collected from a laser scanner, also mounted on the robot, to calculate the shortest path to the robot's target location. The system was powered with a laptop.

While not strictly an object avoidance exercise, clustering the Kinect's point cloud is a technique also employed by Greuter *et al.* [27] to detect 'game elements' (3D objects) for the 2011 Eurobot Challenge. A series of heuristics were applied to the point cloud data to filter noise before the data were clustered. This paper showed the Kinect's performance in classifying the objects to exceed that of a laser scanner, although it noted the Kinect's reduced horizontal field-of-view of 57° compared to the laser scanner's 240°.

Draelos [20] tackled the problem of object avoidance by using the Simultaneous Localisation and Mapping (SLAM) algorithm [7]. This was achieved by using depth data to produce a map of the environment which grows as the robot moves. To do this the Kinect's depth image was manipulated to emulate laser scanner data by only using the values from the middle row (giving the depth readings taken at the exact height of the Kinect). The mapping data was then fed into the Adaptive Monte-Carlo Localisation (AMCL) algorithm [28] to allow path planning. This method guarantees that no observable obstacles will ever be hit, since the path planning will always avoid known walls. However it has the disadvantage that it will only work in a controlled environment where there are no obstacles that fall below the height of the Kinect. It is also a computationally expensive solution and explains the need for tethering the robot to a desktop computer to perform the analysis.

2.3.4 Path Planning

The path planning aspect of the project has been previously investigated by Bruce and Veloso [29] who solved the problem using Rapidly-Exploring Random Trees (RRTs) [30] — a method which expands a search tree from the starting position until it reaches the goal. The paper described an extension of RRTs (which they called ERRTs) which both generalise the technique, allowing more constraints to be placed on the path generated, and give real-time performance in potentially dynamic environments (ones which contain moving obstacles). The system implemented used a camera mounted overhead and a desktop computer to process the results. The resulting system was able to move the robot 40% faster than the previous, reactive object avoidance system.

The Kinect has also been used in real-time path planning by Benavidez and Jamshidi [31], who used an on-board Kinect attached to a laptop to identify objects in the environment, plan routes through it and

control the robot. The data collected was relayed to a connected server which used neural networks to learn appropriate reactions to environmental stimuli and applied this knowledge to provide supervision to the robot. The server did not typically run in real-time, however on occasion the robot would require the server to make a decision on its behalf, in which case the server would reply as quickly as possible.

2.3.5 Target Recognition

The target recognition side of the project is a problem which has to be tackled by almost all practical projects in the field. The approaches employed in the previously discussed works by Mojtahedzadeh [25], Greuter *et al.* [27], and Benavidez and Jamshidi [31] are described below.

Mojtahedzadeh [25] generated a point cloud from the Kinect data, from which points representing the floor were removed. This was then reduced it to a 2D, top-down map of the environment after which it was clustered using the DBSCAN algorithm [3] — a noise resilient, density-based clustering algorithm whereby points in densely packed regions are clustered together. The shape of the clusters could then be analysed, something which was done by using the points to create a convex hull — a polygonal shape which covers all of the points in question.

Greuter *et al.* [27] also generated a point cloud but they ruled out the DBSCAN algorithm as too computationally expensive for their purpose. Instead they reduced the 3D point cloud to a binary 2D image where each pixel indicated whether or not the 3D point mapping to that pixel was present within the specified region of interest (defined by the game field and the known maximum height of the objects on it). This binary representation was then clustered using the run-length encoding (RLE) algorithm [32, pp. 38–61] — a very simple method of extracting groups of nearby pixels. Finally the clusters produced were classified as one of four types of piece based on their detected height and size.

Object recognition was handled by Benavidez and Jamshidi [31] using a combination of the depth and colour data afforded by the Kinect. The targets were detected using the colour data, which was first normalised to remove fluctuations in light intensity, as the targets were physically painted a specific colour to improve detection rates. The depth data was then used to identify open space in front of the robot which it may move into. The depth data were especially well suited to this task since colour data becomes unreliably noisy at long distance.

The DustCart robot [18] avoided the problem by using a hard-coded map for the route with multiple beacons along it to localise itself accurately. Target positions were known on the map so the robot did not have to recognise them, merely stop next to them.

2.3.6 Occupancy Detection

Occupant detection is a less well researched area, although there are some projects of note. Firstly Klomark [33], who investigated using a variety of devices and techniques, stereo vision and structured light methods included, to detect whether a car seat was occupied by a person or not. The focus was only on identification when the camera was in a known, fixed position and aimed directly at the seat. The conclusion of the work was that a stereo vision system was well suited to the task while other, simpler methods performed less well.

Another work, also focusing on occupant detection in the context of car seats, carried out by Marín-Hernández and Devy [34] used stereo vision to identify whether or not an occupant had limbs in certain, critical areas of 3D space around the passenger airbag. Such knowledge allowed decisions to be made about how the airbag was deployed in the event of a collision. This work also used cameras at fixed positions aimed directly at a single seat. The resulting system was able to discern the difference between adults and children as well as their position in relation to the airbag.

A more general piece of research was done by Arras *et al.* [35], where a laser scanner was mounted 30 cm from the ground and used to map a room. From this map the location of people within the room was ascertained via detection of their legs. The environment contained other leg-like objects such as tables, chairs and bins which had to be filtered out. The problem was approached using the AdaBoost technique [36] — a machine learning method which uses many 'weak' classifiers to build a

more powerful final classifier. The resulting system was able to correctly classify people with accuracies in excess of 97%. While not directly similar to this project, it dealt with using a machine learning approach to identify people using depth data of their legs and had to account for noise and differences in orientation.

2.3.7 Marker Detection

For reasons presented in Section 6.1 marker detection was also researched for this project.

Claus and Fitzgibbon [37] created a reliable marker detection system based on a marker consisting of a white square with four black dots inset from the corners. The focus was on creating a highly robust solution insensitive to the scene background and lighting which could also work at long range. Their solution involved using machine learning to create a cascade of classifiers (as employed by the Viola-Jones object detector described in Section 2.2.7) with just two stages: the first running a very simple, computationally inexpensive classifier (a Bayes decision rule) over the entire image to rule out the majority of non-markers, before subjecting the remaining regions to a more complex classifier (nearest neighbour). The result was found to perform with a 95% accuracy under real world, 'difficult' conditions, outperforming an approach not dissimilar to that used by ARToolKit (described below).

Belussi and Hirata [38] approached the problem of marker detection in the context of QR (quick response) codes. These have three square markers (so called 'finder patterns') positioned at the top left, top right and bottom left corners. They used the Viola-Jones object detector (described in Section 2.2.7) to do this and investigated a variety of post-processing techniques to refine the results by removing those where three markers were not detected in the appropriate shape.

The popular ARToolKit library [39, 40] relies heavily upon marker detection to identify objects and their position in the scene, allowing augmented reality applications to superimpose graphics. Its marker detection algorithm is commonly used because it is computationally cheap, accurate (within given ranges) and relatively robust. It detects square markers with thick black borders and a distinctive pattern inside. The marker is stored by the system as a set of 256 feature vectors (distinctive features regarding the structure of the image). The detection then works by thresholding the image, a process whereby all pixels above a certain threshold are set to white while all others are set to black, creating a binary image. Following this the binary image is searched for the feature vectors built up from the marker and areas of the image which exhibit a sufficient number of these features are flagged as potential candidates. These potential regions then have their contours extracted (edges determined based on changing image contrast) and those regions which have contours close in shape to a square are returned as markers. The required closeness in shape is given by a confidence factor where a lower value will follow the marker at longer distances and more oblique angles but also return more false positives. The exact shape of the marker (based on rotation) is also returned. Zhang *et al.* [41] provide an empirical evaluation of a number of marker detection systems including ARToolKit's, showing it to be the fastest but also most prone to false positives. A high frequency of false positive detections is, as discussed in Section 2.2.7, highly undesirable for detection systems working on video streams. This has led to a number of improvements over the stock ARToolKit implementation being suggested, although all of these experience trade-offs in other areas (typically computational expense). Notable among these is Fiala's ARTag system [42] which greatly reduces the false positive rate for only a slight increase in false negatives.

2.4 Alternative Solutions

3D-imaging is required by this project for reasons outlined in Section 1.1. Section 2.1 explains the motivation behind the choice of the Kinect for the 3D camera but does not present the alternative solutions available, instead discussed below.

The 3D-imaging solution requiring the least specialised hardware is a stereo vision setup. In this, two standard cameras are placed a fixed distance apart allowing the depth of those points which are visible from both cameras to be inferred (using much the same theory as structured light projection as

depicted in Figure 2.4, except with two cameras rather than a camera and a projector). The advantage of this configuration is that it uses standard cameras and thus the cost of implementation may be very low. Furthermore as it works on visible light it is able to work in outdoor environments. Disadvantages of the setup are that: the accuracy is heavily limited by the precision with which the distance between cameras can be measured and maintained and the system is far from off-the-shelf, requiring careful construction and calibration. The system's use of visible light also acts to its detriment as it renders the setup sensitive to changes in lighting. The Kinect on the other hand is not significantly more expensive and comes pre-calibrated.

Alternative 3D-imaging solutions involve laser range finding, so called 'time-of-flight' systems where a laser beam is fired at the target and the time until its reflection is seen is measured. Knowing the speed of light the target's distance can then be calculated, providing the time taken can be measured sufficiently accurately. Systems of rotating mirrors allow the depth of thousands of points in an arc around the sensor to be measured every second. These systems have the advantage that their accurate measurement range is far superior to the Kinect, both in terms of close-range targets (which can be measured to a few millimetres) and long-range targets (which can be measured in the order of kilometres, although devices for use in robotics tend to be limited to a few tens of metres). Additionally laser rangefinders are able to work in any lighting condition, outdoors or indoors. They do, however, suffer from several disadvantages, namely the fact that depth data can only be acquired for a single given height, the height at which the laser leaves from. Furthermore laser ranging devices tend to be larger and consume more power than a Kinect. Finally one of the biggest limitations is the cost of the devices: for reference the SICK 180° laser rangefinder used by Arras *et al.* [35] costs around $5500 and the Hokuyo 240° laser rangefinder used by both Greuter *et al.* [27] and Mojtahedzadeh [25] costs around $1000. The Kinect, by comparison, costs $99.

Chapter 3

System Overview

3.1 Module Breakdown

As outlined in Section 1.2 the project is split into four main modules. These were implemented concurrently in a top-down fashion to build a basic system and add functionality iteratively, however they are outlined separately below for clarity. The interaction of these modules is shown in Figure 3.1.

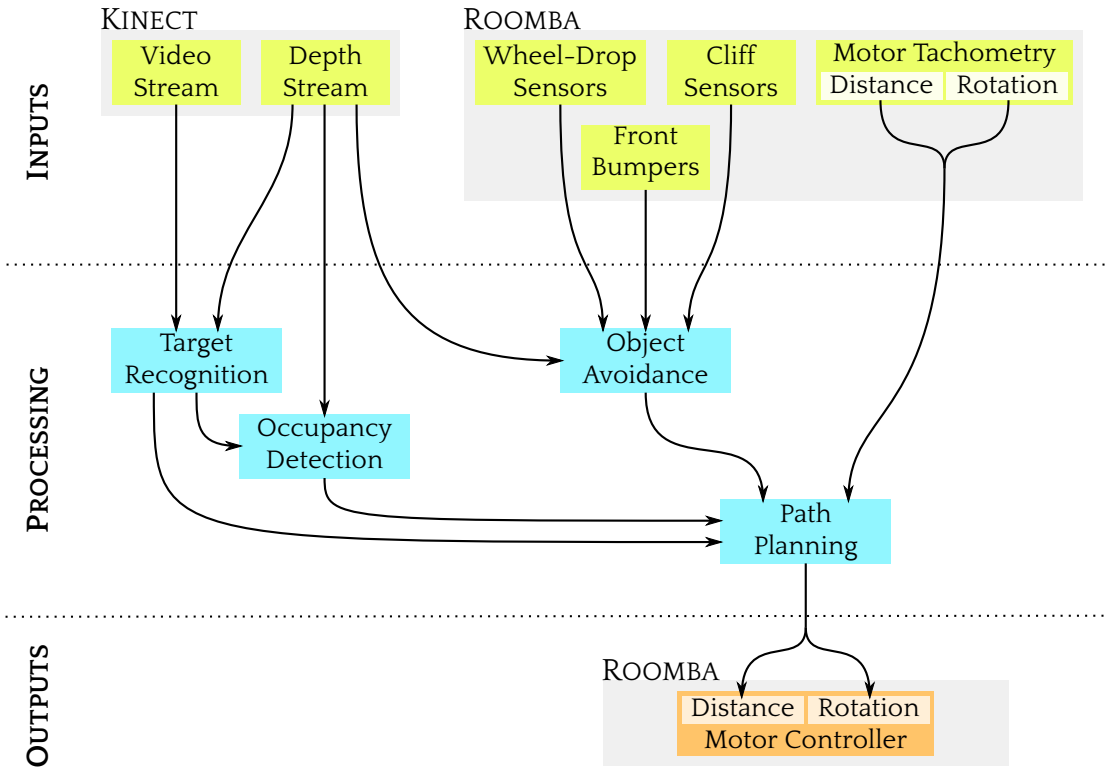


Figure 3.1: The complete system flowchart showing the interaction and data flow between modules.

3.2 Specification

3.2.1 Object Avoidance

The object avoidance module shall:

- Avoid collisions with obstacles in the environment.
- React sufficiently quickly to be able to avoid unexpected events.
- Not produce obstructive behaviour.
- Not be a danger to those around it.

3.2.2 Path Planning

The path planning module shall:

- Explore the environment.
- Create a map so it may revisit places at a later date.
- Stop at tables, allowing them to be serviced.
- Revisit tables at a later date to recheck them.
- Avoid revisiting tables that have been recently serviced.
- Cope with dynamic, constantly changing environments.

3.2.3 Target Recognition

The target recognition module shall:

- Detect tables from a distance.
- Detect tables from any angle of approach.
- Be robust to changes in lighting.
- Calculate the position of detected tables so they may be navigated to.

3.2.4 Occupancy Detection

The occupancy detection module shall:

- Determine whether a given table has at least one occupant (person sitting at it).
- Be robust to discrepancies in position and angle of the table.

Chapter 4

Object Avoidance

4.1 Design

One of the most conventional methods of object avoidance is use of the SLAM algorithm [7] to build a map of the environment and then avoid the walls in that. This has a number of disadvantages however, primarily that it is computationally expensive: in excess of the computational power afforded by the PandaBoard. It also performs best in controlled, static environments where there are no moving objects and obstacles are mostly large, easily detected walls. Another common approach to the problem, especially using the Kinect, is to manipulate the point cloud it produces before eventually using a clustering algorithm to group sets of nearby points together. These clusters are then identified as obstacles and, as their positions relative to the robot are known, evasive action may be taken as necessary. This method was used by a number of the projects discussed in Section 2.3.3.

This project is designed to operate in real world environments which are highly unpredictable in terms of obstacles, some of which will be moving. Any implemented method must therefore be able to cope with such conditions. The task is made easier, however, because unlike much of the related work it is not necessary to discern individual obstacles and avoid them separately, only to identify if an obstacle is present along the current trajectory. This problem was initially split into two sub-problems, avoiding stationary objects in the environment and avoiding moving objects.

4.1.1 Stationary Objects

Stationary object detection was solved by analysing the histogram produced from the Kinect's depth image. This has the advantage of being very computationally efficient — especially important since the system must run in real-time and the faster objects can be detected the more time the robot will have to avoid them. The most basic form of histogram analysis for object detection would rely on using the Kinect's limitation that objects closer than roughly 500 mm are returned as errors (zero values). As a result objects closer than 500 mm can be detected by triggering an object detection when the number of such error readings exceed a certain threshold. Unfortunately due to the noise in depth measurements a large number of error readings will be present in each frame which are unassociated with close-range objects. This means the threshold for detection has to be set quite high, making it a good fail-safe method to stop the robot driving into large and very close obstacles, although a high threshold renders the robot incapable of detecting small objects. The project builds on this method by considering the sum of two regions of the histogram, R_1: 500–700 mm and R_2: 700–900 mm, defined formally as:

$$R_1 = \sum_{i=500}^{700} f(i) \tag{4.1}$$

$$R_2 = \sum_{i=700}^{900} f(i) \tag{4.2}$$

where $f(i)$ gives the number of pixels taking the value of i in a given image. By observing how the number of pixels in each of these regions changes over time it is possible to discern the movement of objects. For example, a transition of pixels from the further region (R_2) to the closer region (R_1) in a short space of time suggests the approach of an obstacle and vice versa. A threshold can be applied to this difference to trigger a response when exceeded, expressed formally as:

$$\text{Object detected} = \begin{cases} 1 & \text{if } (R_1 - R_1' > \tau) \wedge (R_2' - R_2 > \tau) \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

where τ is the threshold applied, R_1 gives the current value of R_1 and R_1' gives the value of R_1 a fixed length of time ago. This method has the advantage that it is far less susceptible to noise than simply checking the error region, meaning the threshold may be set to be more sensitive without increasing the false alarm rate. It also only detects objects moving towards the viewpoint, important because the floor will generate depth readings in this range, however because it will not appear to move towards the robot (even as the robot moves) it will not be detected as an obstacle.

4.1.2 Moving Objects

As the robot will be operating in environments with many moving obstacles it would be advantageous if it could handle these separately. Tracking moving objects would allow the robot to calculate their trajectories and whether or not they cross its own. This would give the robot more time to get out of the way than would be permitted with the method described in Section 4.1.1 and to ignore some objects detected by it which did not pose a threat. Following the successful approach taken by Mojtahedzadeh [25] and Greuter *et al.* [27] to build a point cloud and apply a clustering algorithm to it (Section 2.3.3 provides more background on these projects), it was decided to apply the same method to track moving objects.

Having built the point cloud (as described in Section 2.2.4), in order to extract moving objects from it, it was 'subtracted' from a point cloud built in a previous frame (a fixed length of time ago). This resulted in only the points which had moved a significant distance from their original position since the previous frame. The output of this 'subtraction' is plotted in Figure 4.1.

It is important to note that this method relies on a static viewpoint — if the camera were to move it would appear that all points are moving — however, providing the camera is only moving forwards the point cloud can be easily adjusted to negate movement by simply translating it by the distance moved (which is known). In theory it is also possible to account for rotations, however because it was known the camera would only ever move forwards *or* rotate (see Chapter 5 for details), rotation was ignored and results discarded for 500 ms following a rotation.

Having extracted moving points these could then be clustered. For this the DBSCAN algorithm [3] was used (included in Section A.1). DBSCAN has a number of advantages for this application: the number of clusters in the data does not have to be known beforehand (unlike in algorithms such as k-means clustering); the algorithm can detect (and ignore) noise and the clusters may be any shape (unlike in algorithms such as k-means where clusters are assumed to be Gaussian). The centroid of each of the clusters could then be calculated and compared against the centroid of the cluster in previous frames. Knowing the position of an object at two different times allows calculation of the object's trajectory, whether or not it will impact with the robot and, if it will, where and when (derivation of this is shown in Section B.1).

One of the main drawbacks of the DBSCAN algorithm is that it exhibits an $O(n^2)$ time complexity (methods exist to reduce this to $O(n \log n)$, however for the purposes of this project such methods are impractical). The problem had been circumvented by previously published work [25, 27] via judicious application of computing power. This was something unavailable for this project and it had been hoped that by simply reducing the amount of data being clustered clustering speed could be improved and performance maintained. To reduce the amount of data the image was re-sized to be $1/8^{\text{th}}$ the size in each axis, reducing the number of readings by $1/64^{\text{th}}$ and increasing the distance between readings (in terms of 3D space). Unfortunately this meant that noisy or lost samples had far more of an effect on the overall cluster distribution — thus making it harder for the DBSCAN algorithm to cluster the data consistently in each frame. This introduced a large number of anomalous readings which made it almost impossible to accurately calculate the time to impact and acted to the detriment of the overall

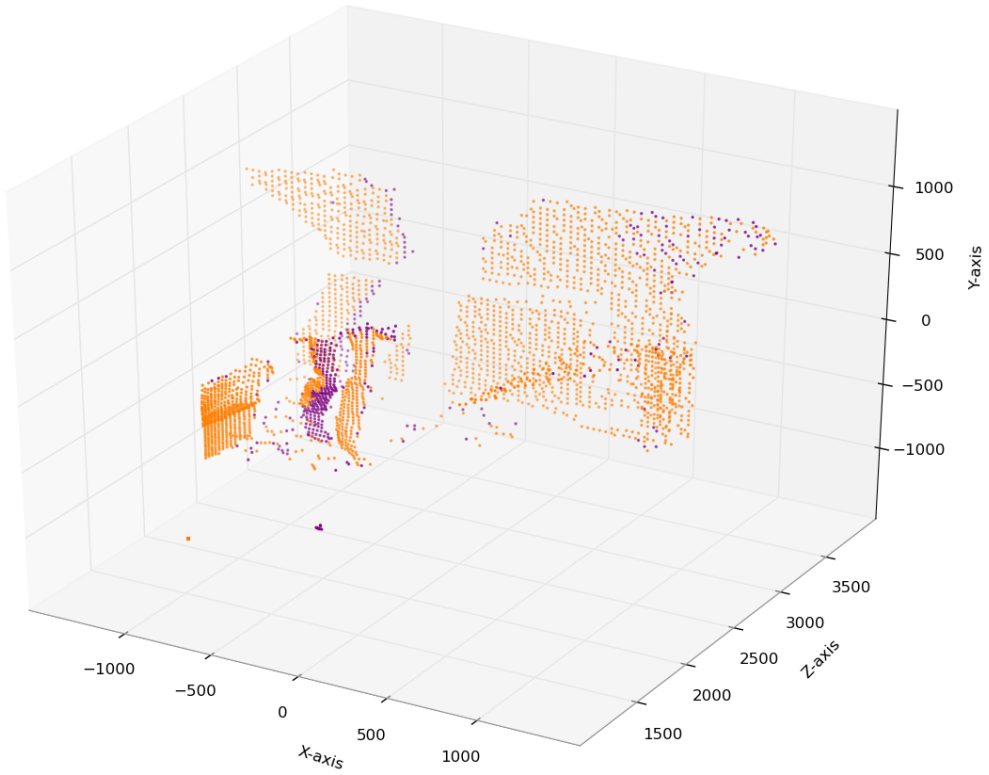


Figure 4.1: A plot of a single frame showing the currently moving points as a person walks past the camera (they are in the process of moving their right leg forwards). The moving points are shown in purple, while the non-moving points are shown in orange. Movement was identified using the method described in Section 4.1.2, where points were considered to have moved if they were more than 50 mm away from their position 500 ms ago.

system. Additionally, as the algorithm was only run on moving points, the more moving points there were in the scene, the longer it took to cluster the data: making the number of frames per second vary wildly. This had an effect on a number of aspects of the system which rely on the frame rate being constant (for example the histogram analysis described in Section 4.1.1) and response times being low (for example the path planning module described in Chapter 5). As a result of this it was concluded that clustering the point cloud to extract moving objects was an inappropriate method in the context of this project.

Instead moving object detection was left to the same system used for stationary object detection (see Section 4.1.1). This meant that a person walking towards the robot would not be picked up until they were within 700 mm of the robot, however it resulted in far more predictable behaviour making it easier for people to judge the robot's intended path and avoid it themselves.

4.2 Implementation

The implemented system consists solely of the method described in Section 4.1.1, not the method described in Section 4.1.2 for reasons described within it.

This uses the depth data provided by the Kinect and necessitates the storage of the histograms generated for use by future frames. A circular buffer was chosen as the structure to store this data as it provides both the theoretically optimal performance and permits an extremely fast implementation in practice. At each frame the histograms generated are stored, as illustrated in Figure 4.2.

In practice the depth data are sub-divided into the left and right halves. The method is then applied

separately to both halves. This has no additional cost in terms of performance and only a very minimal one in terms of memory but allows the robot to discern objects approaching from either the left side, the right side or both simultaneously. This is an important improvement since it allows evasive action to be taken appropriately, for example moving to the left to avoid an object approaching from the right and vice versa.

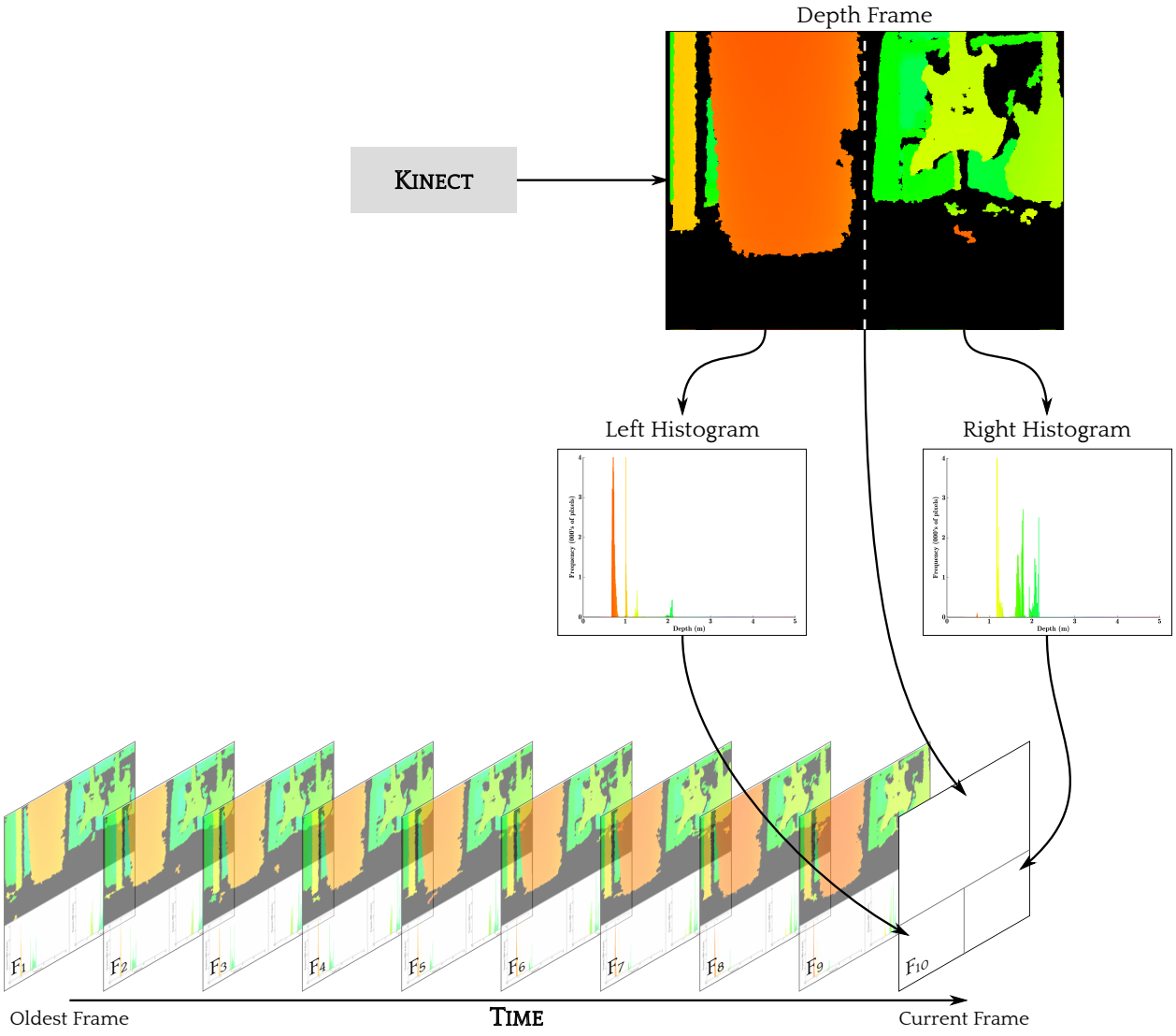


Figure 4.2: A diagram showing the updating of the frame buffer as histograms are generated and stored.

The key parameters affecting the performance of the module are the frame retention: the length of time between the frames which are compared (between R_1 and R'_1 in Equation 4.3) and the threshold applied to the regions (τ in Equation 4.3). The size and positions of the ranges used (R_1 and R_2) are also adjustable: for this project 500–700 mm and 700–900 mm were chosen respectively since they give the robot sufficient time to react and room to manoeuvre without compromising its ability to perform in tight spaces.

The implemented method has a threshold set at 4000 pixels which allows the robot to identify objects which present a minimum of 59 cm^2 in area to the camera (calculated by considering the smallest area which would produce the required number of readings at a distance of 700 mm from the Kinect). This object must then move towards the camera (either due to its own movement or the robot's) within 400 ms (the frame buffer length chosen). For reference a single leg of a standard metal-legged chair typically presents a 95 cm^2 visible area. This is, of course, reduced if it is partly occluded. By always considering the movement of pixels against a point a fixed length of time in the past the robot is not influenced by objects that entered its field of view previously. This effectively removes all objects which are not

currently moving from the robot's consideration. The method is illustrated in Figure 4.3.

The PandaBoard's wireless networking capability was utilised during development to allow visualisation of the robot's progress.

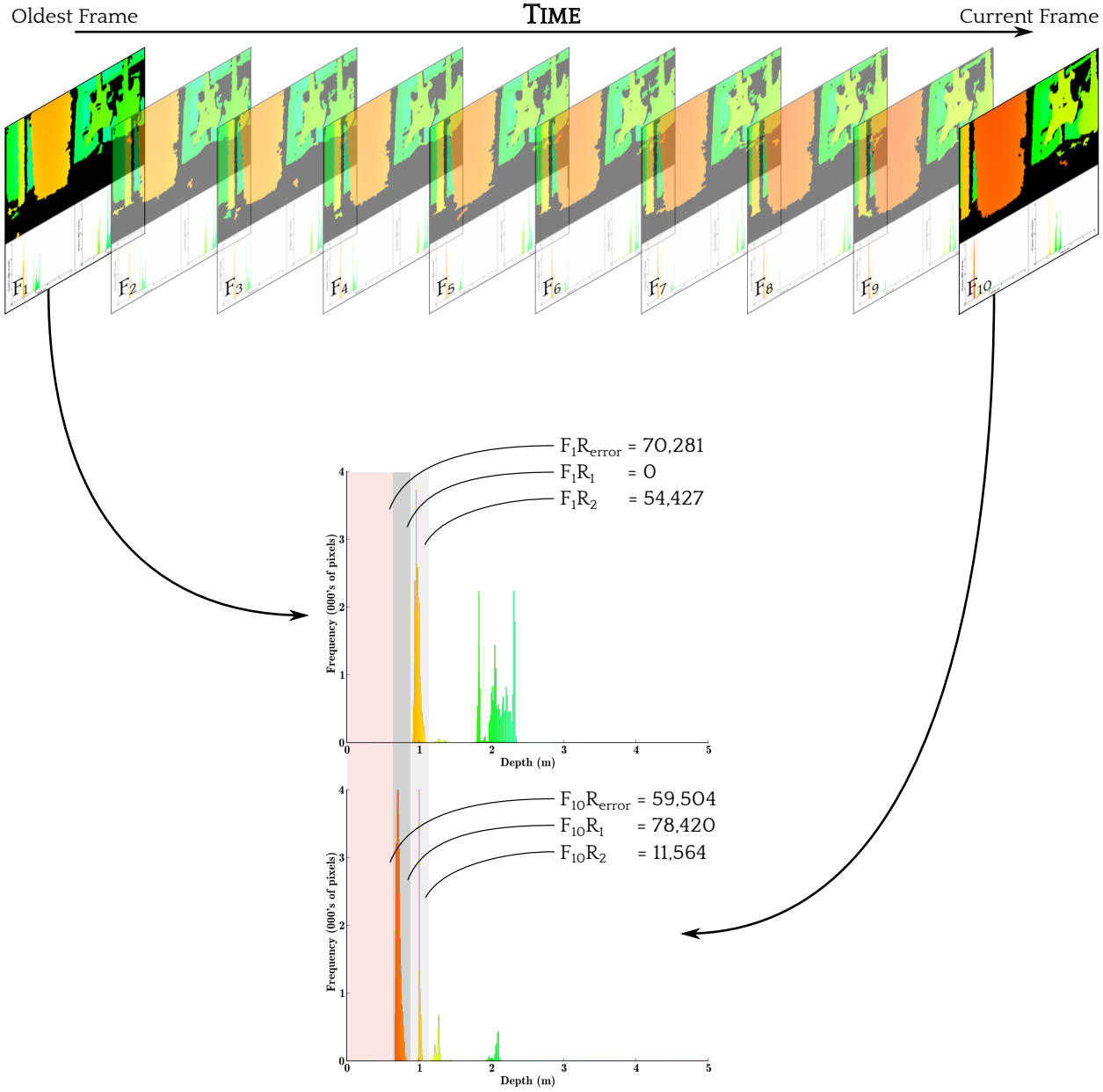


Figure 4.3: A diagram showing the function of the object avoidance method where the newest and oldest frames are extracted from the frame buffer and their histograms considered. To simplify the diagram only the comparison of the left histogram from each is shown.

4.3 Evaluation

4.3.1 Testing

Due to the nature of the object avoidance module it is inherently difficult to evaluate individually because of its reliance on the other modules to react on its obstacle detections. It has, therefore, been evaluated based on its effect on the system when navigating environments with obstacles (an example of a path through which is shown in Figure 5.3).

From a theoretical perspective, the object avoidance module is sensitive enough to be able to detect objects that move towards the camera and have a 59 cm^2 surface area, as mentioned in Section 4.2. Additionally it can detect objects in the Kinect's error range which, assuming these objects are at the maximum 500 mm distance (on the edge of the error range) and no other noise is detected in the frame (unlikely, but it gives a worst-case scenario), this may detect objects that have a surface area of at least 676 cm^2. The required surface area for this detection will decrease as the objects get closer to the camera.

4.3.2 Analysis

The sensitivity of the object avoidance module, as mentioned in Section 4.3.1 enables it to detect most small objects at a distance of 700 mm (the crossover of R_1 and R_2). Additionally it is able to detect large objects that enter the Kinect's error range, however it is unable to detect small objects in this manner due to the large amount of noise produced from the Kinect which would otherwise cause false positive results. This is important to detect obstacles that move into the Kinect's field of view at less than 700 mm. The frame buffer must also be cleared whenever the robot rotates since previous frames would then refer to a different set of 'background' objects. This means that for 400 ms (the frame buffer length chosen) following any turn the module is unable to determine objects moving towards it except if they become close enough to enter the error range. One of the strengths of the module is that it is computationally very cheap, able to easily process 30 frames per second (FPS), the maximum throughput of the Kinect. This allows it to quickly react to obstacles presented to it, vital for safe functionality in dynamic environments, especially those involving interaction with people.

One of the module's downfalls is its tendency to generate false positive results. These do not occur when the camera is stationary, however when mounted on the robot the vibration of the camera as it moves causes generation of false positive results. While this does not happen very often it adversely effects the entire system when it does. From the run recorded in Figure 5.3 four false positives were generated (at nodes 3, 4, 9 and 11) however in all but one of these cases the obstacle was not detected when checking for the second time after a short wait — described in Section 5.1.1. The effects of this could be reduced by stabilising the camera, either through use of a different mounting mechanism or by a software image stabilisation approach. Alternatively the robot's speed could be reduced, although this may increase the obstructive behaviour of the robot. For this project it was not deemed necessary to address the issue since the solutions would slow the robot down in terms of computational or physical speed.

Additionally due to its mounting position, the Kinect is closer to the ground than might be ideal. This causes the entire floor to be unreadable in terms of depth so the bottom half of the frame is populated by error readings. This problem could be solved by angling the Kinect upwards, which was tried, however it then misses small objects on the ground such as protruding feet of tables and chairs. An alternative solution is to raise the mounting position of the Kinect. In experimentation it was shown that the Kinect only needs to be raised by an additional 100 mm in order to be able to correctly read the depth of the floor, resulting in large improvements in terms of the error readings generated, although in doing so an increase in the vibrations experienced by the camera was also observed. The adjustment, therefore, can be thought of as improving the module's ability to detect objects in the error range (since there would be less zero-readings from the floor that would mask the presence of such objects) at the expense of its ability to detect moving objects (since the camera would move around more, causing the threshold for object detection to have to be raised). As such the adjustment was not deemed necessary for this project.

The granularity with which the module produces detections could also be improved by subdividing

the image further than just left and right. This is possible without impacting the performance of the module as the image would still only be fully passed through once, only the counts would be recorded by separate histograms. This would allow obstacles detected to the far left or right of the depth image to be avoided by small movements, however the module would become more impacted by noise since a patch of noise would proportionally cover more of each area, potentially fully obscuring a region resulting in an object being falsely detected in the Kinect's error range.

Chapter 5

Path Planning

5.1 Design

The goal of path planning in this project is to explore the environment in order to find tables and avoid colliding with known obstacles or becoming stuck. Ideally the rate of exploration should decrease with time as the environment becomes fully mapped, in favour of simply visiting known tables.

The problem of path planning has as many different solutions as there are projects that require it — it is entirely dependent on the problem at hand and each individual project's requirements. The examples discussed in Section 2.3.4 are a handful of approaches, with the use of Rapidly-Exploring Random Trees (RRTs) being a common component in many solutions (as used by Bruce and Veloso [29]). RRTs require knowledge of the map beforehand though (Bruce and Veloso used a camera mounted looking down on the arena), making them unsuitable for use in this project. Benavidez and Jamshidi [31] used a path planning technique focusing on exploration by building a map of the environment and using a neural network to process environmental stimuli. This is not a viable method for this project however as it made use of an external server to build the neural network and even then it was not designed to run in real-time.

The solution implemented by this project is a light-weight approach inspired by the examples mentioned above. One of the things that allows this solution to be simpler than some is that the robot does not have a specific target position it must reach, only a higher level goal for it to maximise litter collection while minimising obstructive behaviour. The approach involves building a map as a topological representation of the environment: a graph with nodes connected by a series of edges. Nodes then represent reachable points and edges viable movements. A graph representation such as this is more intuitive for a robot to work with and is easier to build up on the fly than a full, metric representation that a human might prefer where all walls and obstacles are explicitly marked out. The map itself will be built using the tachometry data provided by the Roomba's wheel driving motors using dead reckoning (described in Section 2.2.6). As this method is subject to cumulative errors it will be necessary to re-localise periodically using the tables encountered — the position of which can be worked out relative to the current viewpoint rather than the map (described in Section 6.1). The map is only required to discover tables and viable paths between them, so it need only be sufficiently accurate to allow the robot to pass close enough to the table that it may detect it using the target recognition module described in Section 6.1.

The path planning module is split into three distinct types of action it may take. These actions are combined to achieve the high level goal mentioned above and are as follows:

1. Random Movement — A random movement a fixed distance in any direction, designed to explore the environment. This allows the robot to locate new tables as well as new paths between known locations which may be faster than existing paths.

2. Random Node — A movement to a random known node, designed to focus exploration around known parts of the map. This allows both linking the current location to existing regions of the map and exploring around known parts of the map. This is an action inspired by RRTs which

explore in a top-down manner, at first making large crude movements to discover the structure of the map before making small precise movements from these later on to fill in the map.

3. Greedy Node — A movement to a random known table, designed to exploit knowledge of the map and achieve the fundamental goal of collecting litter.

These three actions are described in more detail in Sections 5.1.1, 5.1.2 and 5.1.3 respectively. The probability that these actions are chosen will change as more of the environment is explored, allowing greater exploitation of the located tables. It is important that exploration is not stopped prematurely however, as this could lead to inadequate servicing of all tables. The most intuitive way of slowing down exploration rate and increasing exploitation rate is to ensure $\Pr(\text{Greedy Node}) \propto |T|$ where T is the set of all discovered tables. Likewise the number of random node actions, which visit already known nodes in an attempt to explore around them (roughly emulating RRTs), should increase as more of the environment becomes mapped. This may be achieved by ensuring $\Pr(\text{Random Node}) \propto |G|$ where G is the set of all discovered nodes (including tables). Finally once tables have been visited they should not be visited again immediately since it will be unlikely that any further benefit will be found: if the table was empty it will either still be empty or have new occupants who are unlikely to immediately produce litter and if the table was full it will either now be empty or still full, in which case it is unlikely the occupants have immediately produced more litter. In order to accomplish this behaviour all the identified table nodes are given a weight which represents how likely they are to be chosen as targets. This weight increases over time and is reset when the node is visited.

Taking into account all these constraints, the probabilities associated with each action are worked out in practice using the following equations:

$$\Pr(\text{Greedy Node}) = \left(\frac{\sum_{t \in T} w_t}{100 \cdot |T|}\right)\left(1 - \frac{1}{|T| + 1}\right) \tag{5.1}$$

$$\Pr(\text{Random Node}) = \big(1 - \Pr(\text{Greedy Node})\big)\left(0.1\left(1 - \frac{1}{|G| + 1}\right)\right) \tag{5.2}$$

$$\Pr(\text{Random Movement}) = 1 - \Pr(\text{Random Node}) \tag{5.3}$$

where G is the set of nodes (including tables), T is the set of tables and w_t gives the weight of a table t where weights have a minimum value of 0 (when just serviced) and a maximum value of 100. The rate at which the weight increases is something which can be adjusted based on the number of tables and size of the environment.

For this project the robot has a maximum move distance of 2000 mm for Random Movements and moves at 200 mm/s. The weight of tables is increased by 1 every 6 seconds (up to the maximum of 100). While the weight itself may increase slowly, the fact that updates happen several times a minute result in just less than a 50% probability of a Greedy Node action having been taken after two minutes on a map with two tables both starting at weight zero. This relationship is further explained and graphed in Section C.1.

5.1.1 Random Movement

Random Movements will make up the majority of the actions early on and are intended to explore the map, setting off at a random angle for a maximum distance of 2000 mm — therefore generating a single rotation followed by a single movement. If an object is located during a random movement it will terminate the action and set the next random movement to be at right angles to the object. The

angle to rotate is sampled from a Normal distribution.

Algorithm 1: Random Movement Procedure. Initially $\mu = 0,\ \ \sigma = 20°$.

Rotate θ, where $\theta \sim \mathcal{N}(\mu,\ \sigma^2)$.
Move 2000 mm or until an object is detected.
if *an object is detected* **then**
  Stop, wait 1 second and recheck.
  if *the object continues to be detected on the left* **then**
    $\mu \leftarrow 90°$
    Stop.
  else if *the object continues to be detected on the right* **then**
    $\mu \leftarrow -90°$
    Stop.
  else
    Continue.
  end
else
  $\mu \leftarrow 0°$
end

5.1.2 Random Node

Random Node movements will become increasingly common as the map size grows (tending to a 10% probability). They act to keep exploration focused around the existing nodes as the map grows. As with random movements they consist of a single rotation followed by a movement. The angle to rotate is calculated by considering the robot's current orientation as a vector and then calculating the angle from it to the vector linking the current node and the target node. Upon encountering an object, as with the Random Movement action (see Section 5.1.1), the action will simply be terminated. It is also necessary to set μ (used by Algorithm 1) appropriately to ensure a Random Movement will not be generated at the next update which attempts to drive into this object again.

Algorithm 2: Random Node Procedure, given the current position $c = (x_c,\ y_c)$, current orientation γ and node position $n = (x_n,\ y_n)$. This may also change the value of μ used by Algorithm 1.

Rotate θ to face the node, where $\theta = \arccos\big((x_n - x_c)\sin(\gamma) + (y_n - y_c)\cos(\gamma)\big)$.
Move d to reach the node or until an object is detected, where $d = \sqrt{(x_n - x_c)^2 + (y_n - y_c)^2}$.
if *an object is detected* **then**
  Stop, wait 1 second and recheck.
  if *the object continues to be detected on the left* **then**
    $\mu \leftarrow 90°$
    Stop.
  else if *the object continues to be detected on the right* **then**
    $\mu \leftarrow -90°$
    Stop.
  else
    Continue.
  end
else
  $\mu \leftarrow 0°$
end

5.1.3 Greedy Node

Greedy Node movements will become more common as the number of discovered tables grows. They also become more likely the longer tables have gone unserviced (for reasons described in Section 5.1). Unlike the other two types of action (detailed in Sections 5.1.1 and 5.1.2) Greedy Node actions should not

just be abandoned if an obstacle is found. If they were, due to the fact the probability of these actions can become overwhelmingly large over time, the robot could constantly be ordered to move to a table, find an obstacle directly in front of itself and give up — becoming stuck. The other actions do not suffer from this problem as they are indifferent about the actual position moved to, so long as some movement is achieved. Thus, the robot should initially try to travel directly to the table, however if obstacles are encountered it should turn back to the closest node in the map and then follow the shortest path through the map to the target. This allows the use of the map's edges which were successfully traversed at some point in the past. If a further obstacle is found the path is recalculated. Since, on sparse maps, it is possible the new path will reuse the obstructed edge a limit must be put on the number of path recalculations before the robot gives up.

Algorithm 3: Greedy Node Procedure, given the current position $c = (x_c,\ y_c)$, current orientation γ, node position $n = (x_n,\ y_n)$ and node orientation ρ.

Rotate θ to face the node, where $\theta = \arccos\big((x_n - x_c)\sin(\gamma) + (y_n - y_c)\cos(\gamma)\big)$.
Move d to reach the node or until an object is detected, where $d = \sqrt{(x_n - x_c)^2 + (y_n - y_c)^2}$.
if *an object is detected* **then**
  Stop, wait 1 second and recheck.
  if *the object continues to be detected* **then**
    Calculate the closest node n' behind the current position (the algorithm to do which is described in Section A.4).
    Repeat this algorithm (Algorithm 3) to move to n'.
    Calculate the shortest path from n' to n using Dijkstra's algorithm (included in Section A.2).
    Repeat this algorithm (Algorithm 3) to follow the path, moving from node to node.
  else
    Continue.
  end
end
Rotate $\rho - (\gamma + \theta)$ to face the target.

5.2 Implementation

As described in Section 5.1 the path planning module revolves around building up an internal map of the environment consisting of nodes and edges. The module generates a type of action, as described in Sections 5.1.1, 5.1.2 and 5.1.3, according to a set of probabilities (given in Equations 5.1, 5.2 and 5.3). Each of these actions is made up of three possible movements: a rotation followed by a forward movement followed by another rotation. This combination allows the robot to move from any point and orientation on the map to any other point and orientation, providing no obstacles are hit. In the case of the Random Movement (Section 5.1.1) and Random Node (Section 5.1.2) actions the third rotation is not used as the finishing orientation is unimportant. Nodes are placed on the map every time an action is completed and linked to the previous node. The distances used in the map are built up using the dead reckoning technique (described in Section 2.2.6).

Target recognition is used to re-localise the map since the target positions can be determined relative to the current position rather than using the map, as described in Section 6.1 and illustrated in Figure 6.2. Providing tables are encountered on a regular basis this will prevent errors from accumulating. Figure 5.1 shows the expected actions of the path planning module.

The nodes in the map which are tables are stored and have separate weights. These weights are increased over time (to a maximum of 100) and reset when the table is visited, as described in Section 5.1.

None of the libraries used by this project permit the generation of normally distributed, pseudo-random numbers. In order to provide this functionality the Box-Muller transform was used (included in Section A.3). Normally distributed random numbers were required for the implementation of Algorithm

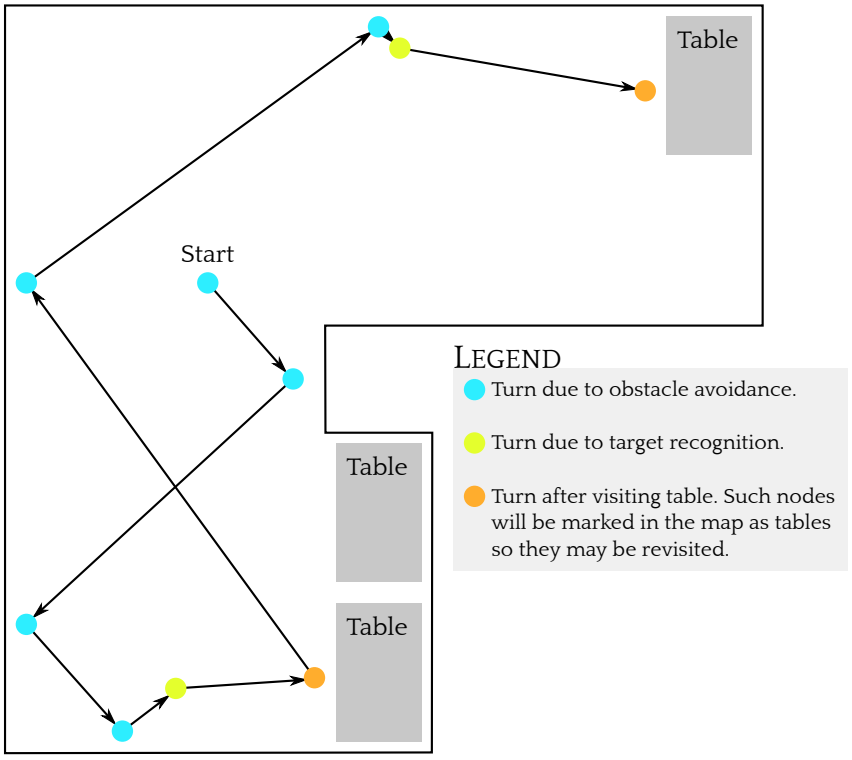


Figure 5.1: The expected behaviour of the path planning system with it randomly exploring (blue nodes) until a table is found (green nodes) at which point it will service the table (orange nodes) before continuing.

1, for which randomness is introduced to avoid potentially repeating paths (for example if the robot turned exactly 90° at each obstacle, four well placed obstacles could result in it travelling in a square between them indefinitely).

5.3 Evaluation

5.3.1 Testing

The path planning module's performance can be evaluated by examining the map it generates, which is saved to a text file at the end of a successful run. A simple simulator was created to allow the path planner's behaviour to be observed without having to wait for the robot to trace the path, greatly increasing the speed with which it was developed. This simulator overrides the wheel controls, emulating the output of the motor tachometry without moving the robot. The sensors are still used as normal so items may be moved towards the robot to prompt simulated object avoidance routines and targets may be shown to prompt simulated target recognition routines.

Figures 5.2 and 5.3 show typical maps produced by the path planner as it explores. Figure 5.2 was generated by the simulator and shows movement through an environment with no obstacles. A marker was presented to the simulator midway through the run. Figure 5.3 was generated from a real run of the robot and involves it exploring an open area of floor in a computer laboratory. The environment contained large items such as desks and chairs as well as smaller items such as bags and litter.

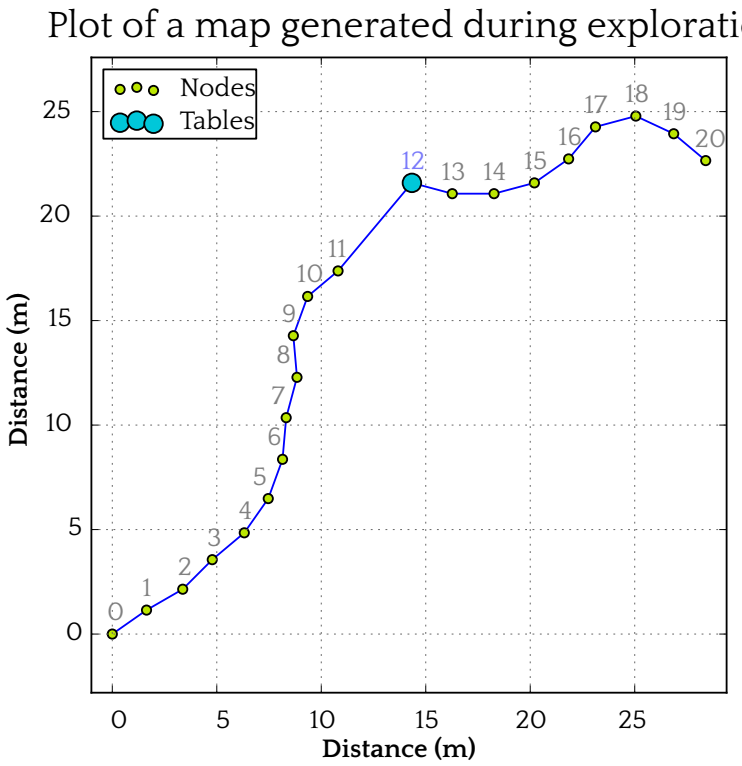


Figure 5.2: A map produced by the path planning module through an unconstrained environment (i.e. no obstacles were encountered). This map was produced by the simulator.

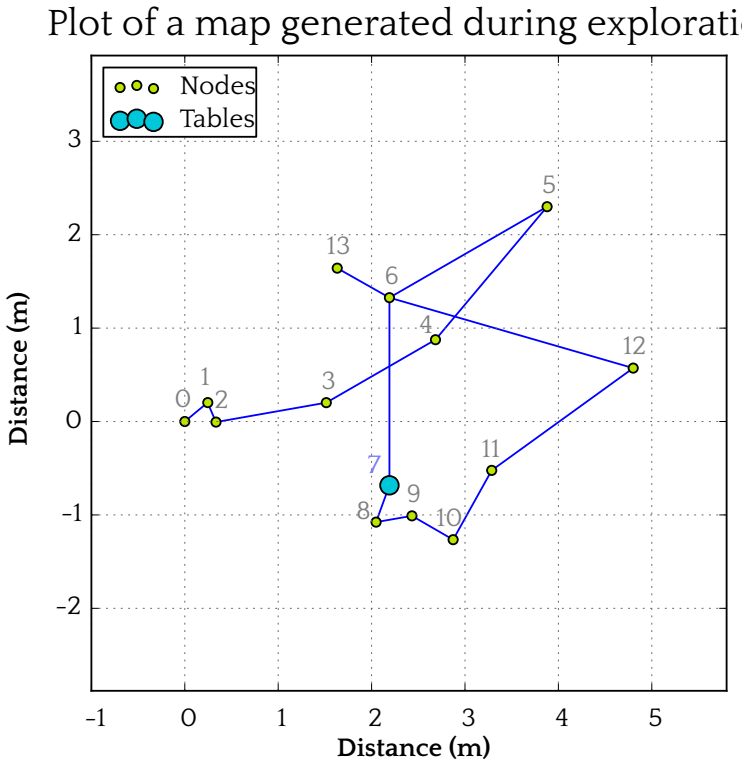


Figure 5.3: A map produced by the path planning module from real world performance (i.e. with obstacles). This map was produced from a real run executed by the robot.

5.3.2 Analysis

Figure 5.2, a map of a simulated run through an environment without obstacles, shows heavy emphasis on the Random Movement actions (described in Section 5.1.1) for exploration. The addition of the normally distributed noise to the angle of travel after each movement cause the robot to 'meander' through the environment without significantly impacting on the overall direction travelled. This allows the robot to explore the environment in greater detail than if it had just driven straight until it met an obstacle as well as reducing the reliance on starting angle. Node 12 shows where the marker (described in Chapter 6) was presented to the robot to represent a table. The marker was then removed from view after being detected. Having serviced the table the robot then turns to its right (to avoid hitting the table) and continues exploration. No nodes are revisited in this run.

Figure 5.3, a map of a real run through an office environment, shows a more cautious approach with frequent adjustments from the object avoidance module. The robot initially was redirected by a false positive object detection from the object avoidance module, however recovered and was able to complete full exploratory moves soon after. A target was correctly observed following node 6. Having serviced the table the robot unfortunately set off towards the table (which did not have any chairs) and had to navigate around its legs, which it did successfully. Node 6 was revisited from node 12, showing an example of a random node action (described in Section 5.1.2). It is also possible that the movement from node 5 to 6 was intended to be a random node action to node 1, which falls in line with it, however an obstacle was encountered and it changed path.

The path planning module is a simple solution, compared to other available methods which are computationally expensive, and it is capable of exploring and mapping the environment in a sophisticated manner. It is able to alter its level of exploration over time to exploit the knowledge gained about the map. One of the main weaknesses of the approach, however, is its inability to be recover from external actions, for example if it were unexpectedly moved or rotated. This would fail to be accounted for and the robot would continue building the map as if it had not been moved. This would affect the performance when it tried to revisit part of the map built before the movement, instead causing it to revisit somewhere different. This is a particular problem in the case of tables which would continue to increase in weight as they went unserviced and the robot would be unable to correctly find them. This is also an event which is likely to happen considering the bin is designed to interact with people, however solving the problem is not straightforward, requiring a continuous localisation procedure, akin to that used by the SLAM algorithm [7] — a much more computationally expensive method. The robot's tachometry would record movement undertaken due to external action if the wheels were on the ground and maintained traction throughout, as would be the case for small external force, however if a person were to knock into the robot with force or deliberately move it, the robot's map would become incorrect. A further weakness of this method is its inability to exactly pinpoint the current position on the map due to the errors accrued from dead reckoning. In the case of this project this is not a problem as only approximate distances need to be known.

The module is sufficient for the needs of this project, and without external actions it is able to track position and generate movements as required, however its inability to recover from external actions would need to be addressed before deployment to a highly crowded environment.

Chapter 6

Target Recognition

6.1 Design

The goal of the target recognition module is to identify tables as the robot navigates the environment. The task is made additionally complex as the tables to be identified could be at any scale, rotation or distance. In order to increase the generality of the table detector (so it could be easily applied to other types of tables rather than just one) as well as its accuracy it was decided to apply markers to tables. These could be of any design as long as they were easily detectable and not intrusive (i.e. excessively large). This would, in theory, allow the markers to be applied to any table in any environment and still have the robot achieve the same performance. An additional bonus to using markers is that they are of a known design and size. This will give the robot known landmarks to help re-localise its map with, a necessity due to the use of dead reckoning leading to cumulative errors (as described in Section 5.1). In order to use the markers as landmarks the robot need not know the positions of markers in advance (indeed, to do so would defeat the aim of a general purpose solution), only their size. Knowing their size allows the robot to calculate its distance from them and realign itself with them (e.g. to be directly facing the marker at a set distance away), thus re-localising the map using the table. The markers must be detected using the Kinect's conventional video stream rather than its depth video stream since it would be difficult if not impossible to discern a marker attached to a table leg with depth information alone.

There are a number of approaches to marker detection, discussed in Section 2.3.7. Each of these systems has its own advantages and disadvantages. The system implemented by Claus and Fitzgibbon [37], designed for marker detection in 'difficult' environments, used a large marker — while they never explicitly stated the marker's dimensions, they compared it against other methods using 20×20 cm markers and images of the system suggest their marker was at least that size. With these markers the system was shown to work at up to 10 m and with targets rotated up to 75° around their vertical axis (from the camera's perspective). The system ran at 8.3 frames per second (FPS). The system implemented by Belussi and Hirata [38], designed to detect markers on QR codes, ran at 8.0 FPS. The detection range was dependent on the QR code size (which was not fixed) and the relationship was never formally evaluated. Finally, the ARToolKit [40] method [39], designed for use in augmented reality systems, has been shown to work with 20×20 cm targets at up to 3 m [43], although the official documentation states that a target of that size will only have an effective range of 1.27 m [44]. Additionally the markers continued to be detected when rotated at angles up to 60° around their vertical axis (from the camera's perspective) [41]. The system ran at 76.3 FPS [41].

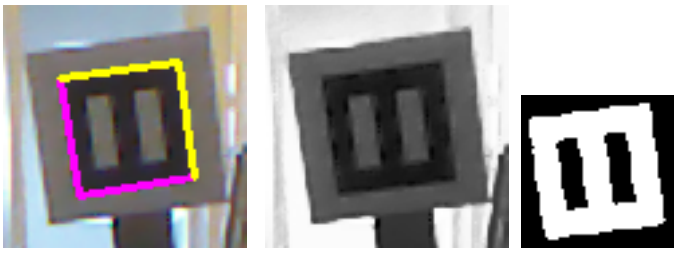
One common theme across all methods was the marker design utilising only black and white colours, aiding in detection and improving the result of thresholding the image (as black and white are the least ambiguous colours to split). For this reason purely black and white markers are also used in this project.

The ARToolKit solution is attractive on account of its speed, and, while it is accurate for short-range detection, it is incapable of detecting items at long range — making it impractical for use in this project. Instead it was chosen to use a cascade of classifiers as utilised by Claus and Fitzgibbon and Belussi and Hirata, shown in both examples to be slower than the ARToolKit method but far more robust to changes

in lighting and capable of tracking at a longer range. Specifically the Viola-Jones object detector, as used by Belussi and Hirata and described in Section 2.2.7, was chosen due to it being an effective, fast, general purpose solution which can be easily customised by adjusting its parameters. For this project it is important that markers be tracked at relatively long-range (in the order of metres) but are not overly large — the 20×20 cm markers as used by the described works are far too large to be inconspicuously attached to tables in the environment.

One disadvantage of using the Viola-Jones object detector is that it only works on square windows of the image (sliding the window back and forth row by row) on different scales (as described in Section 2.2.7). This means that detected markers are always a square region of the image regardless of the orientation of the marker (for example a marker rotated 45° from the camera would appear to be half as wide as it is tall but the marker detector would still return a square detection covering non-marker pixels on the left and right). Additionally because the windows are scaled up by a percentage amount each time it is likely that the detected region will also contain some non-marker pixels at the top and bottom. Thus the Viola-Jones method is only capable of determining approximate areas in the image which contain markers: it is unable to actually determine the shape or orientation of those markers. Due to this a post-processing step had to be performed to extract the four marker's corners from the detected region, important so the orientation and distance of the marker could be deduced (the need for which is described above).

Before the Viola-Jones object detector is applied to the image it is converted to grayscale and its histogram is normalised. This has the effect of boosting the contrast of the image and somewhat (although not completely) reducing the effect that lighting will have on the image. Following the Viola-Jones detection, approximate regions containing markers are returned and each of these is extracted from the image. The extracted region is then thresholded: pixels with a value less than 64 are converted to white while all others are converted to black (this in effect inverts the colours of the marker, making white regions represent the outer edge of the marker and black the background). The result of this conversion can be seen in Figure 6.1c. Noise is then removed from the thresholded image via use of connected-component labelling — a fast technique which operates on binary images, scanning through the image and grouping together white pixels which are next to each other. This results in the labelling of continuous regions so, in the context of the marker detection, the marker will be in one large region (assuming all its associated pixels are touching) and any background noise will be in a series of smaller regions. All regions which have an area below a given threshold are then removed leaving only the large region(s) making up the marker. Finally the corners are extracted from this cleaned image by, for each corner, taking the white pixel which is closest (diagonally) to it. Providing all noise has been successfully removed and the marker's edge has been preserved these points will correspond to the marker's true corners. The stages of this process are shown in Figure 6.1.



(a) Colour frame (with marker outlined).

(b) Normalised grayscale frame.

(c) Thresholded marker region.

Figure 6.1: Three images from the marker detection process (described in Section 6.1). The image was first captured in colour by the camera (a) and converted to grayscale (b). The Viola-Jones object detector was then applied to the grayscale image to extract the approximate marker region (the area shown in (c)). A binary threshold was then applied to this and noise removed (c) before the corner points were extracted — connected and shown as coloured lines in (a).

As well as extracting the marker's corners to calculate its position, the corner extraction step provides an additional layer of verification for the object detector's output because the positions of the four corners are constrained geometrically. While the corner extraction does rely on the object detector to supply approximate regions, due to the extremely low computational cost incurred by running it,

it allows some parameters in the object detector to be adjusted in favour of speed at the cost of an increase in false positives. Furthermore the object detector frequently 'misses' frames, detecting an object in one frame of a video and then missing it in the subsequent frame. This behaviour can be remedied by re-running the corner extractor on the previously detected region in the new frame. Providing the object has not moved significantly the corner extractor will be able to find and extract the corners again, in effect classifying the marker without the object detector's input. The object detector can then be seen as bootstrapping the system: finding regions which are then searched in all subsequent frames until a marker fails to be detected (which may be at the next frame).

Having extracted the four corner points their depth is read using the Kinect's depth camera. The corresponding 3D points for the four corners are then calculated (as described in Section 2.2.4) and the width and height of the marker derived (taking into account differences in depth caused by rotation). If the four points form a square acceptably close to the true size of the marker (edge lengths of 50 mm $\pm$ 20 mm) the detection is classified as a marker and the position in 3D space and orientation are returned. Given the four points TL, TR, BL and BR representing the 3D positions of the marker top left, top right, bottom left and bottom right corners respectively, the marker position P is calculated as the average position of the corners:

$$P = \frac{TL + TR + BL + BR}{4} \tag{6.1}$$

and the marker orientation θ may be calculated as follows:

$$\theta = -\arctan\left(\frac{(TR_X - TL_X) + (BR_X - BL_X)}{(TL_Z - TR_Z) + (BL_Z - BR_Z)}\right) \tag{6.2}$$

The position P' which is q units directly in front of P can then be calculated by:

$$P' = (P_X + q\sin\theta\,,\ \ 0\,,\ \ P_Z + q\cos\theta) \tag{6.3}$$

The arrangement of these points with respect to the camera is shown in Figure 6.2.

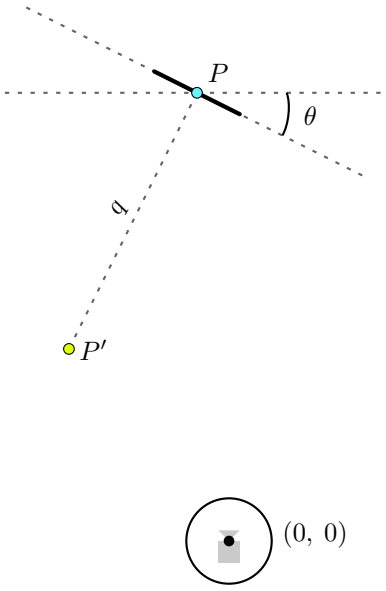


Figure 6.2: Diagram illustrating the relationship between the marker position P at orientation θ and the target position P'.

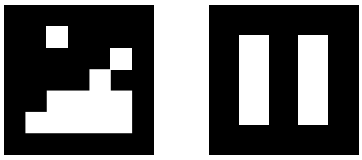
This output may then be used by the Greedy Node procedure (Algorithm 3) to move to P', as described in Section 5.1.3, allowing the robot to navigate to a fixed distance directly in front of the marker. Additionally the point P (and by extension P') is relative to the robot's current position, not the internal map, allowing re-localisation of the map.

6.2 Implementation

The marker detection requires the Kinect's RGB colour video stream, however because past frames will never need to be reexamined it is not necessary to store these in the frame buffer.

An important consideration is the use of the depth image in conjunction with the colour image to find the depth of individual pixels. While this is theoretically sound, the Kinect's RGB and depth cameras are mounted 25 mm apart (shown in Figure 2.3) so the same pixel in both will correspond to different physical points in 3D space. However, by knowing the distance between the two cameras and the depth of the points (as given by the depth camera) it is possible to transform the depth image to appear as if it were taken from the RGB camera's perspective. This works by calculating the 3D coordinates of all pixels in the original depth image (a technique described in Section 2.2.4) and then projecting these points back through a virtual camera positioned 25 mm to the left, giving a new depth image where each pixel correlates exactly to the same pixel in the RGB image. While all the pixels correlate exactly the image is not necessarily identical to the true image which would be observed if taken from that position. This is because there could be unseen obstacles, occluded from one viewpoint which are not so from the other, something which would not be portrayed in the transformed image. Thus the method works for small distances between real and virtual cameras, as used here, where such discrepancies will be both small and rare; however for large distances between viewpoints it will fail to work as the discrepancies become overwhelming large. This virtual viewpoint transformation is provided by the OpenNI library [6].

The design of the markers was also something which had to be carefully considered. As described in Section 6.1 black and white markers were chosen for their ease of detection when viewed in grayscale. The final marker design is shown in Figure 6.3b, however other marker designs were considered. The first one used for prototyping, shown in Figure 6.3a, was one taken from the ARTag library [42, 45]. Once the system had been prototyped, time was invested in designing a more distinctive marker which would be detectable at longer range. The marker shown in Figure 6.3b was chosen as it features a thick black border, necessary for the corner extraction procedure, and two large white stripes which are easily representable using the Haar-like features employed by the Viola-Jones object detector (pictured in Figure 2.13).



(a) Marker design 1. (b) Marker design 2.

Figure 6.3: The two different marker designs, with (b) being the final design.

Having decided upon the marker design training data had to be produced. One of the Viola-Jones object detector's greatest weaknesses is its requirement for large amounts of training data (as discussed in Section 2.2.7), however for simple markers which do not change in appearance it is possible to automatically generate training data. This is done by taking the marker and placing it in a background image while also warping it to emulate its appearance if it were rotated in a 3D environment. Using this procedure thousands of positive samples can be generated at varying rotations, scales and brightnesses. The bounding rectangle containing the marker is also saved so that it may be extracted from the image to train the classifier on (however because the image will have been distorted it will no longer be a perfect square so elements of the background will also be extracted). For this project the marker deformations were limited as follows:

- Rotation around X (pitch) — $0° \pm 14.3°$.
- Rotation around Y (yaw) — $0° \pm 57.3°$.
- Rotation around Z (roll) — $0° \pm 28.6°$.
- Maximum value alteration — $0\% \pm 39\%$ (i.e. black pixels may become up to 39% the brightness of white pixels and vice versa, in effect moving all pixel values toward grey).

These values were chosen based on likely presentations of the marker. Because the markers will be placed as upright as possible it is unlikely the robot will encounter a marker with a significant Z-axis rotation (so that the pattern would be at an angle), and even less likely that it will encounter a marker with X-axis rotation (so that it would be facing the ceiling or floor). However, because the robot may approach the markers from any angle it is very likely it will encounter Y-axis rotation (where the marker would not be directly facing the camera). Likewise it is also important to make the detector as invariant to lighting as possible to accommodate for a range of environments. An example from the automatically generated training images is shown in Section D.1. 5000 positive and 5000 negative images were used, requiring a total of 10,000 negative images of which 5000 underwent the procedure above to produce the positive images. For this the negative data sets used for training face detectors were collected, the largest among these by far being the data set generated by Wu *et al.* [46] for their work in the field. The tool used to automatically generate the positive images is part of the OpenCV library [2].

Only a subset of the Haar-like features available, shown in Figure 2.13, was used by the detector. From the full feature set there are three distinct sets of features used:

- BASIC — Features (a), (e) and (f).
- CORE — Features (a), (e), (f), (b), (g) and (h).
- ALL — All features depicted.

The performance associated with each of the feature sets is examined in Section 6.3.

The OpenCV [2] implementation of the Viola-Jones algorithm offers three different types of boosting, Discrete AdaBoost, Gentle AdaBoost and Real AdaBoost. The Gentle AdaBoost technique was chosen for this project because of research done by Lienhart *et al.* [16] which suggested that it outperforms the other two in terms of accuracy (being at least no worse in any of the tests) and offers considerable speed increases over them.

6.3 Evaluation

6.3.1 Testing

The Viola-Jones object detector used has many tunable parameters to increase both its speed and accuracy. Due to the somewhat unusual nature of this task, detecting very distinct easily definable markers rather than more difficult to determine objects such as faces, the effect these parameters had was investigated. The addition of the corner extraction procedure allows adjustments to be made in favour of a reduction in computational cost or a decrease in the false negative rate at the expense of the false positive rate.

The parameters tested are listed below in the order tested and with their default value: the value used for all other tests. The parameters are described in detail in Section 2.2.7 and, for convenience, briefly recapped below:

Classifier Size
The size positive image samples are reduced to before searching for Haar-like features during training. Larger values allow more features to be considered, increasing training time.
Default value: 16×16 pixels

Classifier Training Size
The number of positive and negative samples used during the training. Larger values increase training time.
Default value: 3000

Classifier Splits
Describes the actual structure of the individual classifiers. A value of 1 generates each classifier as a single-level decision tree (also known as a stump) whereby the classification occurs based on a

single threshold (identical to the behaviour shown in Equation 2.6). A value of 2 generates a two-level decision tree where each node's classification is based on a single threshold. A diagram illustrating this structure is presented in Section D.2. Larger values increase the complexity of each classifier but allow relationships between features to be learnt.
Default value: 3

Classifier Mode
The set of Haar-like features used in the classifier. This may be either BASIC, CORE or ALL — the corresponding features for each are described in Section 6.2. A larger feature set increases training time but allows more complex features to be represented.
Default value: BASIC

Detector Scaling Factor
The rate at which the detector's window is scaled up. A value of 1.2 means that with each full pass through the image the window is made 120% the current size. This process is then repeated until the window is larger than the image in any dimension. Larger values cause the window to grow more quickly, potentially missing objects in between scales but decreasing computational cost.
Default value: 1.2

Detector Minimum Neighbour Count
The minimum number of neighbouring windows which must detect the object to register a true detection. Larger values reduce the false positive rate while increasing the false negative rate.
Default value: 1

Marker Style
The type of marker used. This is one of the images from Figure 6.3.
Default value: Figure 6.3b

Marker Angle
The marker's yaw or angle around the Y-axis (relative to the camera). Positive values indicate clockwise rotation.
Default value: 0°

Marker Size
The size of the physical marker across the black edge. While impractical for this project, markers up to 200×200 mm in size were tested to allow direct comparison of the system to other work in the field [37].
Default value: 50×50 mm

Two different types of tests were carried out.

For the classifier and detector parameters the resulting object detector was run through an unseen set of 1000 images, each of which contained a single marker in the same style as those used for training. Correct detections were recorded when the object detector detected an object which fell within ±25% of the true position. All other detections were recorded as incorrect. To remain consistent with other work in the field [17] correct classifications are presented as a percentage of total possible correct classifications (1000) and incorrect classifications are presented as the absolute number. The total of correct and incorrect classifications for each run need not total 1000 since for any given image the detector could detect zero or more markers. As this was a test of just the object detector without the corner extractor, a higher percentage correct is generally more desirable than lower numbers incorrect. The results to these tests are shown on Page 47.

The second type of test performed examined real world performance in terms of detection range — the single most important factor for this project's application. In these tests the marker was placed facing the Kinect at a given distance away. The entire marker detector system (object detector and corner extractor) was then run and the number of frames in which the marker was correctly extracted was recorded. These tests were carried out from 750 mm to 2500 mm in 250 mm increments. Tests were attempted at 500 mm, however as this is on the edge of the Kinect's close-range (described in Section 2.2.1) the corner extractor tends to reject any detected markers on the basis that it cannot correctly determine their size. For the variable marker size experiment distances up to 6500 mm were tested. The results to these tests are shown on Pages 48–52.

Classifier Size (pixels)	Correct Classifications	Incorrect Classifications	FPS
12×12	55.9%	78	17.2
16×16	87.0%	223	17.2
20×20	89.8%	283	17.2

Table 6.1: The results of classifier size on object detector accuracy. Results are out of 1000 images.

Training Size	Correct Classifications	Incorrect Classifications	FPS
1000	86.9%	257	17.2
3000	87.0%	223	17.2
5000	87.4%	194	17.2

Table 6.2: The results of training set size on object detector accuracy. Results are out of 1000 images.

Classifier Splits	Correct Classifications	Incorrect Classifications	FPS
1	82.8%	119	17.2
2	86.0%	211	17.2
3	87.0%	223	17.2

Table 6.3: The results of classifier splits on object detector accuracy. Results are out of 1000 images.

Feature Set	Correct Classifications	Incorrect Classifications	FPS
BASIC	87.0%	223	17.2
CORE	84.8%	152	17.2
ALL	80.8%	133	17.2

Table 6.4: The results of the feature set used on object detector accuracy. Results are out of 1000 images.

Scaling Factor	Correct Classifications	Incorrect Classifications	FPS
1.1	94.0%	462	16.1
1.2	87.0%	223	17.2
1.3	77.9%	113	23.5

Table 6.5: The results of scaling factor on object detector accuracy. Results are out of 1000 images.

Minimum Neighbours	Correct Classifications	Incorrect Classifications	FPS
1	87.0%	223	17.2
2	78.9%	55	17.2
3	72.9%	19	17.2

Table 6.6: The results of minimum neighbouring detections on object detector accuracy. Results are out of 1000 images.

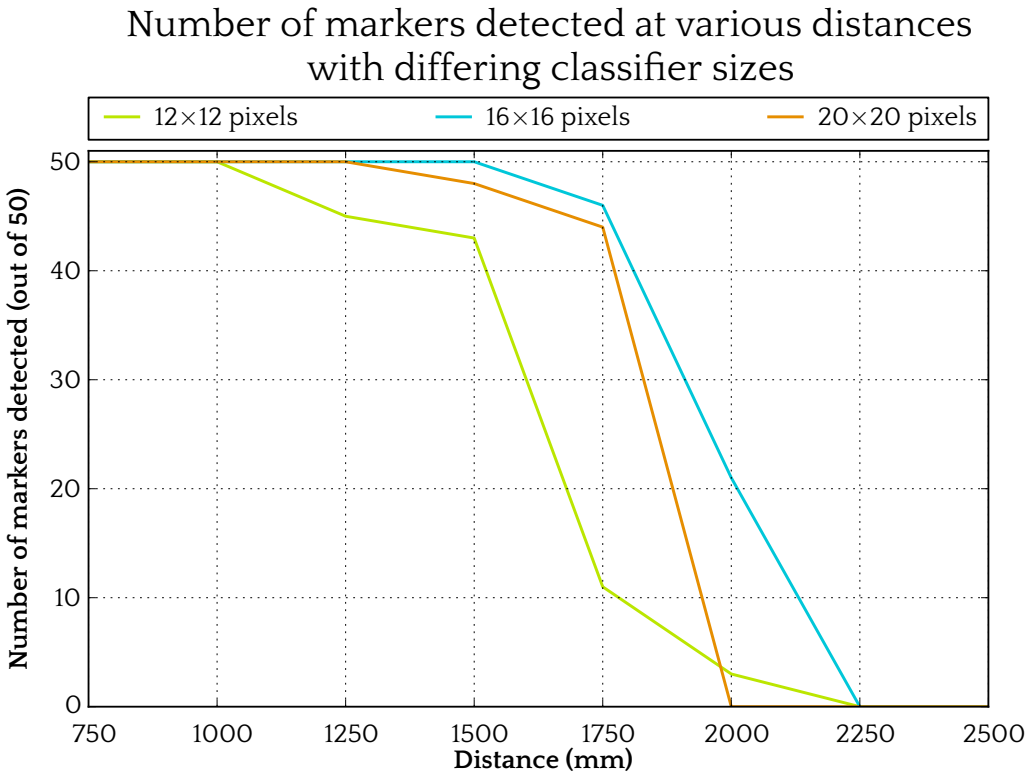


Figure 6.4: The results of classifier size on marker extractor range.

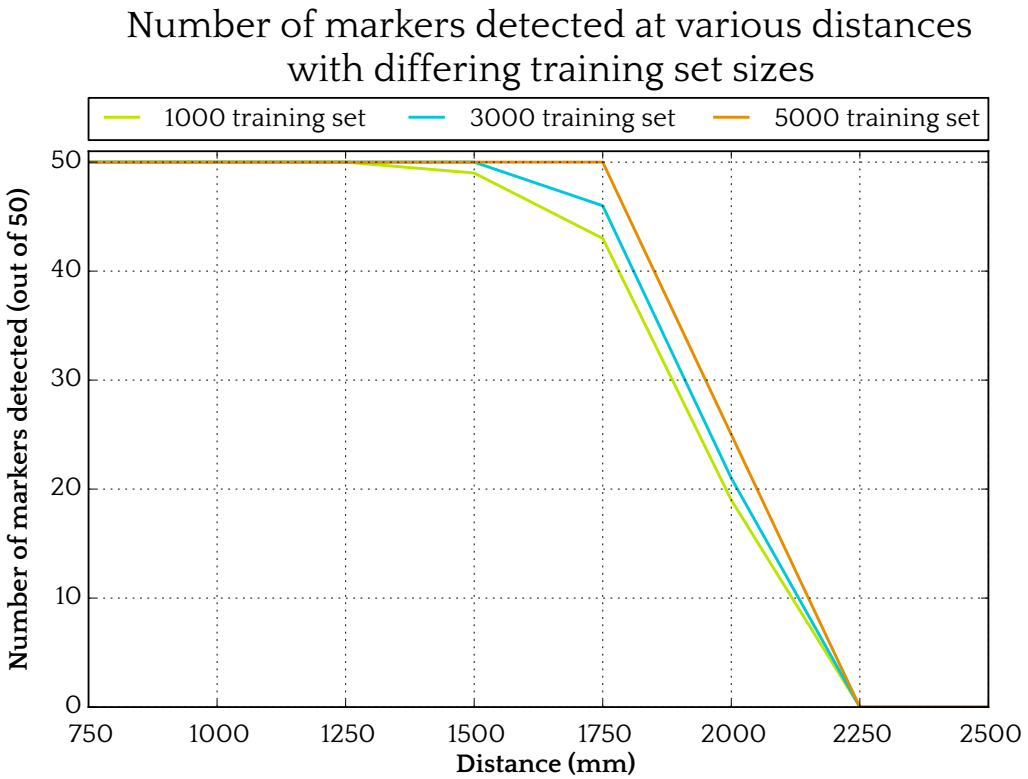


Figure 6.5: The results of training set size on marker extractor range.

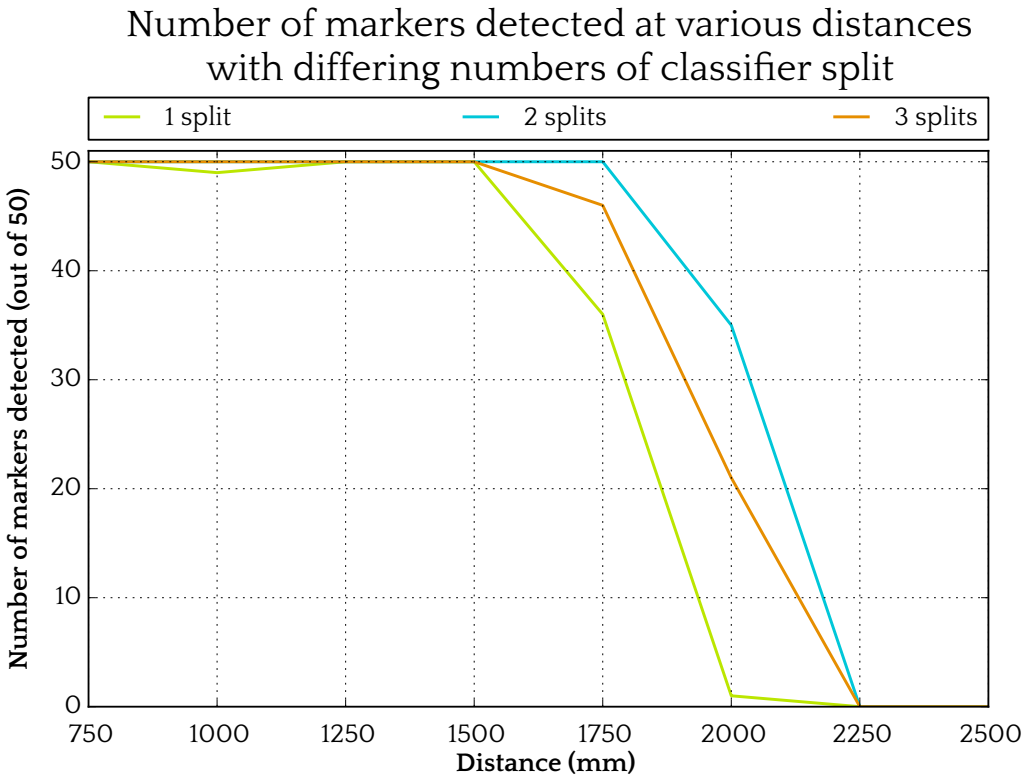


Figure 6.6: The results of classifier splits on marker extractor range.

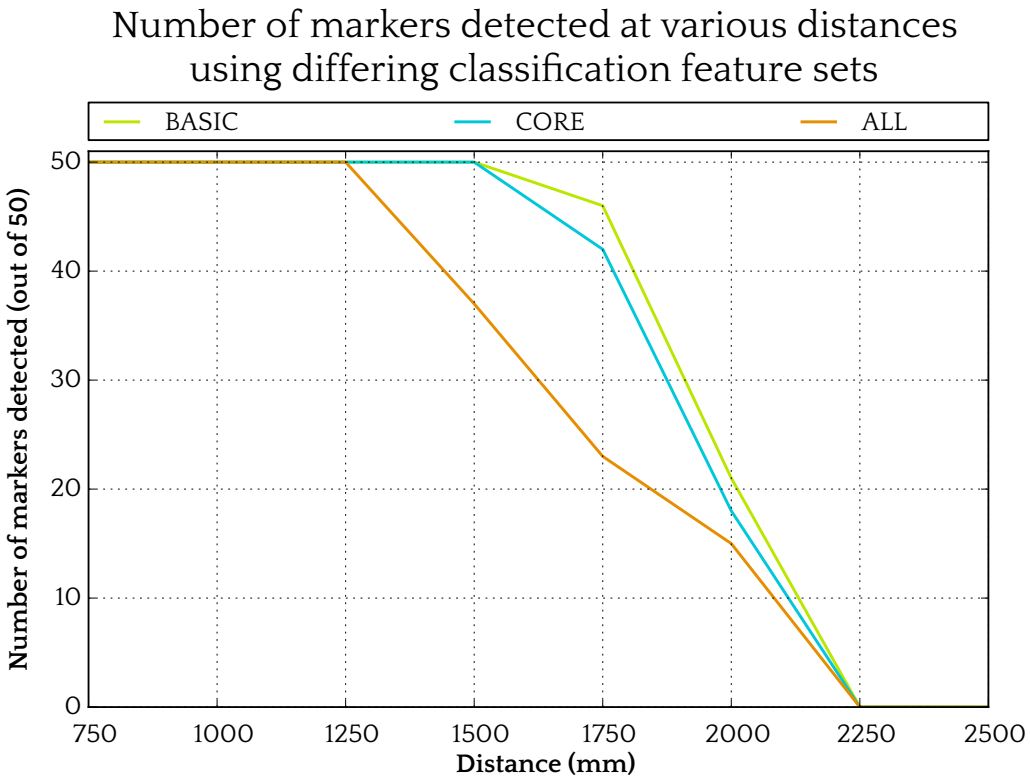


Figure 6.7: The results of classifier feature sets on marker extractor range.

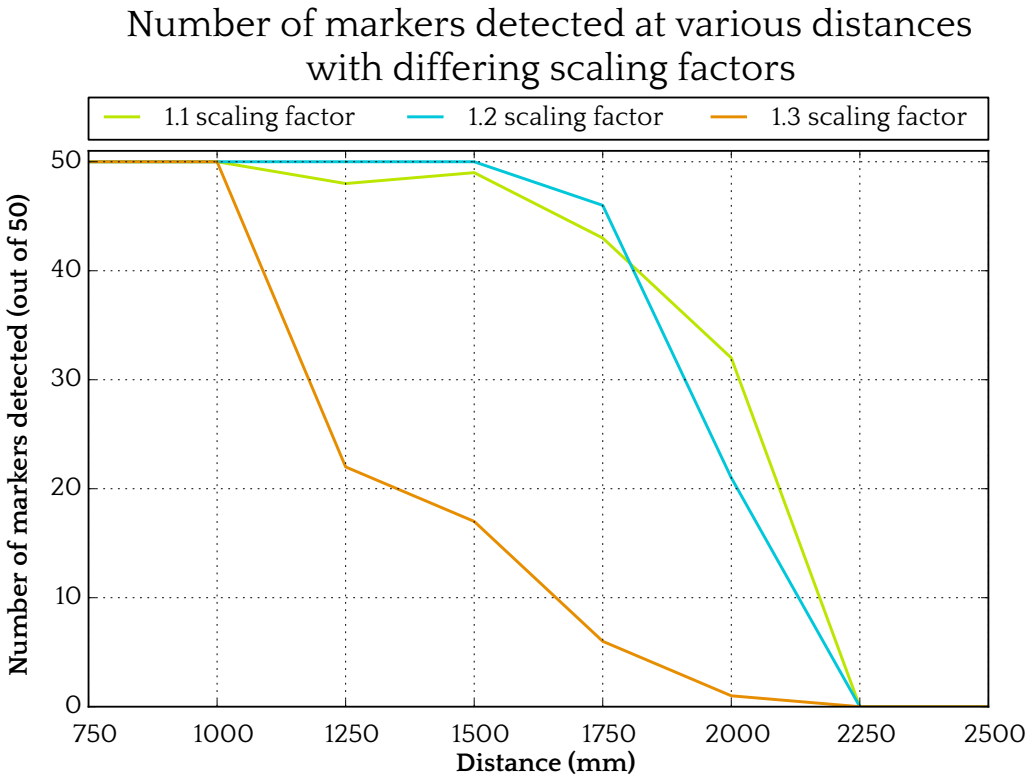


Figure 6.8: The results of detector scaling factor on marker extractor range.

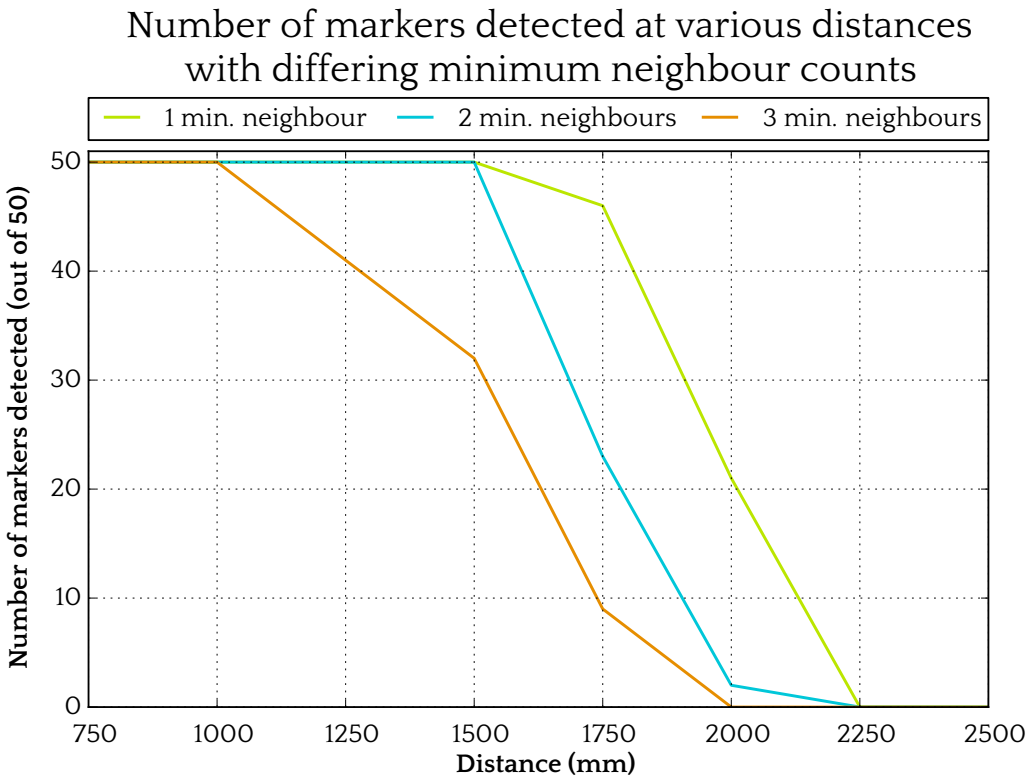


Figure 6.9: The results of detector minimum neighbour count on marker extractor range.

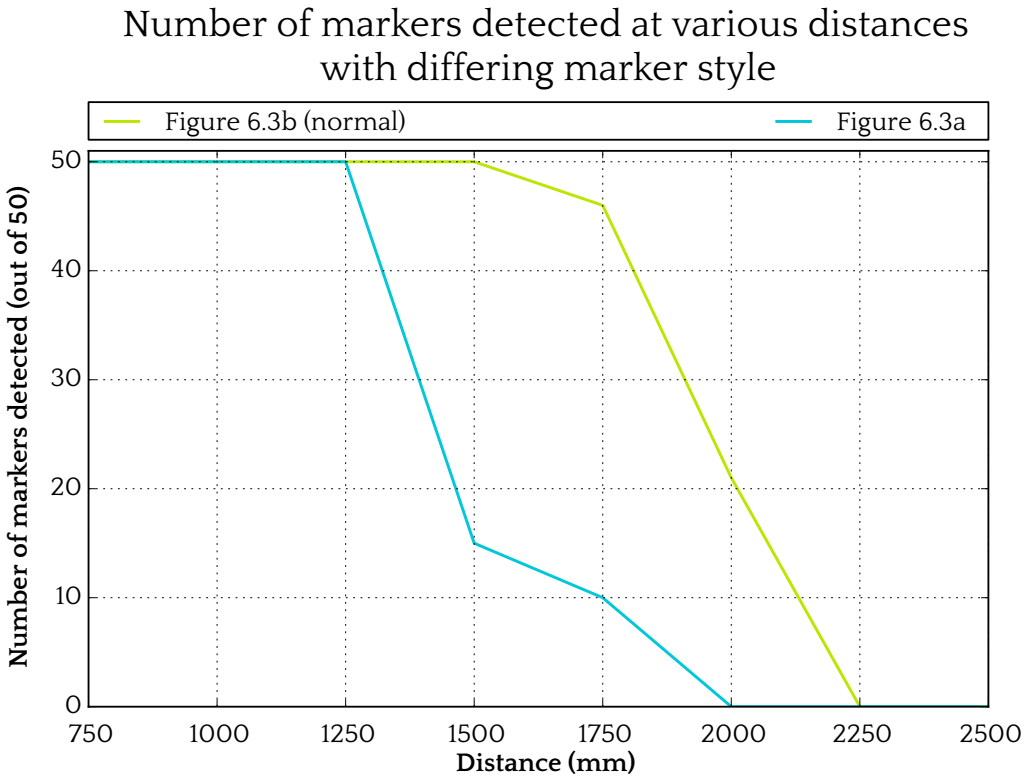


Figure 6.10: The results of marker styles on marker extractor range.

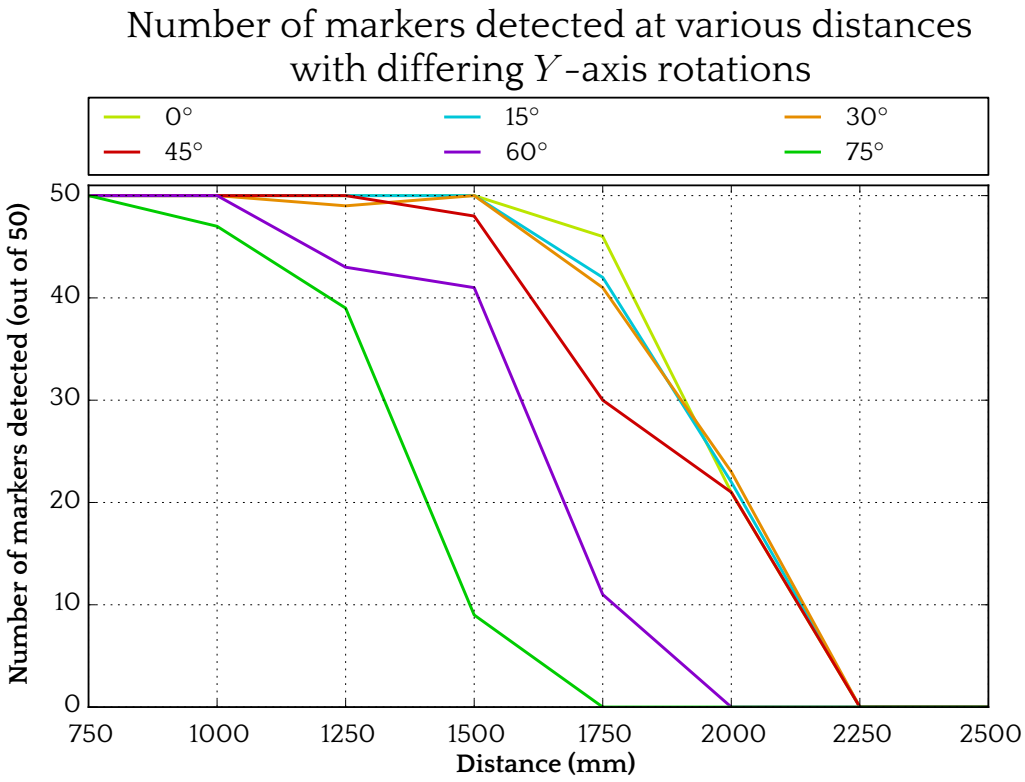


Figure 6.11: The results of marker angle on marker extractor range.

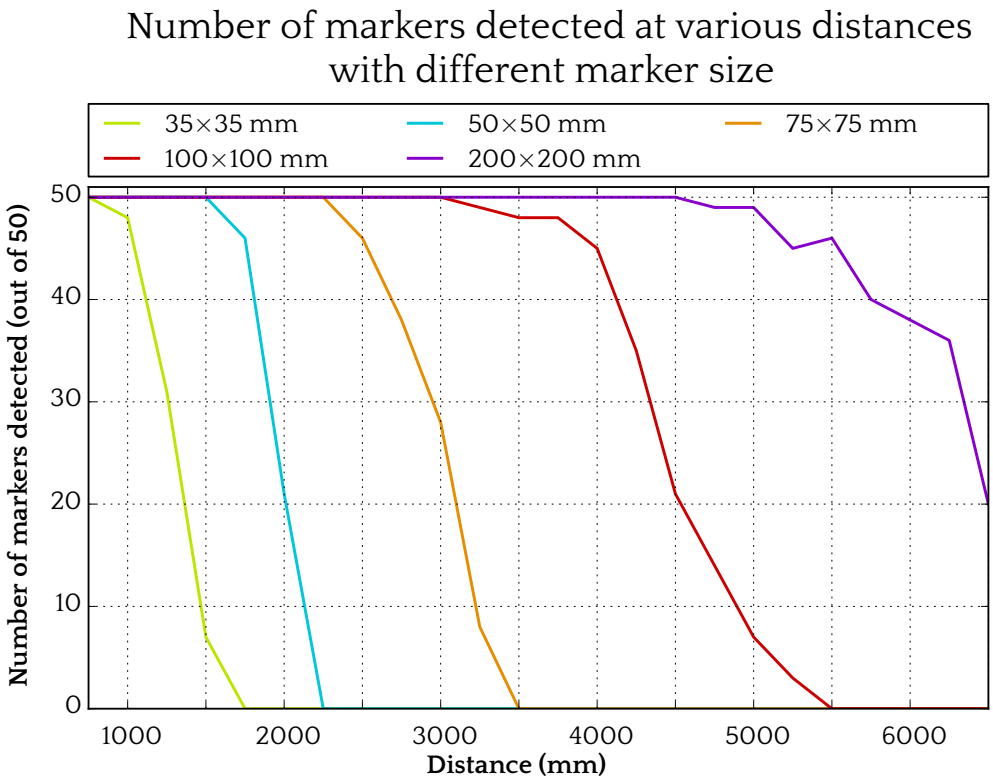


Figure 6.12: The results of marker size on marker extractor range. Note that this graph is plotted with a different x-axis range to the others.

6.3.2 Analysis

The results of the experiments are discussed in the order they are presented above (Pages 47–52).

The results of the classifier performance on varying sizes of classifier window, presented in Table 6.1, show that larger size classifiers increase the number of correct classifications but also lead to more incorrect classifications. There is a large increase in performance when increasing classifier size from 12×12 to 16×16 pixels with a much more modest increase from 16×16 to 20×20 pixels.
This result is backed up by the real world tests, shown in Figure 6.4 where the 12×12 pixel window performs worse throughout and 16×16 and 20×20 are comparable until the far range where the 16×16 pixel classifier does better. Larger size classifiers also show a much harsher cutoff in terms of range, with the 20×20 pixel classifier failing completely in the 1750–2000 mm range while the 12×12 pixel classifier continues to function, albeit poorly, throughout this range. This is likely caused by the fact that classifiers are unable to detect objects smaller than themselves so larger classifier sizes miss long-range objects while smaller classifiers can find them. However increasing classifier size also increases the accuracy with which it functions. The 16×16 classifier appears to be the best compromise from those tested.

Experimenting with increasing the training set size, the results to which are shown in Table 6.2, appears to increase the correct classifications while decreasing the incorrect classifications, leading to all-round 'better' classifiers.
This is backed up by the real world performance, illustrated in Figure 6.5, where larger training sets improve the effective range, although there is not a significantly large difference between the three.

Increasing the number of classifier splits increases the complexity of each individual 'weak' classifier. The results of testing the final 'strong' classifier, shown in Table 6.3, suggest that increasing the complexity of the classifiers increases both the number of correct and incorrect classifications with a large improvement from 1–2 splits and a smaller improvement from 2–3.
The effect was tested in practice, shown in Figure 6.6, and does not reveal a great difference in performance, although 1 split suffers towards the end of the testing range. 1 split performs the worst throughout and 2 the best. The decrease in practical performance by the 3 split classifier is likely due to it overfitting the training data so that it works on the synthetic samples it was trained and tested on but does not transfer as well to actual camera footage.

Testing the use of different feature sets to build the 'weak' classifiers, presented in Table 6.4, shows that adding more possible features for each classifier to evaluate decrease their accuracy as well as the number of incorrect classifications.
This is borne out in the practical experiment, the results to which are shown in Figure 6.7, where the ALL feature set does considerably worse than the other two, overfitting the data. CORE and BASIC exhibit comparable performance with BASIC (the smallest feature set) doing slightly better. This is to be expected considering the design of the marker which does not feature any diagonal lines (the main addition by the ALL feature set to CORE). Furthermore the most prominent feature in the marker design are the two vertical white lines, which will be picked up by the BASIC feature set alone. The additional horizontal lines the CORE feature set is able to detect are not as prominent and harder to detect, especially at small scale, so likely dilute the classifiers by focusing on these substandard features.

Larger scaling factors cause the detector window to scale more quickly, reducing the number of passes made through the image. The effect of this on classifier performance is investigated in Table 6.5 where larger scaling factors are shown to considerably reduce the number of both correct and incorrect classifications. Additionally, larger scaling factors improve performance — the only parameter tested found to do so.
In the real world range experiment, presented in Figure 6.8, the 1.3 scale factor is shown to substantially reduce the effective range of the marker extractor. Meanwhile 1.1 and 1.2 scaling factors show similar performance but with a scaling factor of 1.2 offering a moderate speed improvement. Smaller scaling factors are shown to have a more rapid cutoff in terms of range, similar to that observed by increasing classifier size.

Table 6.6 investigates the effect of the required number of minimum neighbours for a detection to register. It finds that increasing the minimum number of neighbouring detections reduces the object detector accuracy but also greatly reduces the number of incorrect classifications.
In the practical experiment, shown in Figure 6.9, the corner extractor rejects any additional false pos-

itives generated as a result of a lower minimum neighbour count so discounting the increase in false positives associated with lower minimum neighbour counts. Thus the only effect increasing the minimum neighbour count has is to reduce the effective range of the marker extractor.

Investigated in Figure 6.10, the new marker style is shown to greatly improve long-range performance, likely due to the fact it is a simpler design. The design can be far better represented with the Haar-like features and so it is more easily distinguished at long distance when much of the intricate detail is lost.

Figure 6.11 shows that changing the marker's Y-axis rotation appears to have little influence on the detectable range up to 45°, after which performance degrades. This ability to accurately detect markers at large angles is a very useful characteristic in the context of this project where the marker may be approached from any angle. Furthermore, despite the decrease in performance past 45°, markers can still be correctly detected up to 75°, providing they are close enough.

From a theoretical viewpoint it is expected that doubling the physical size of a marker will double the range it can be seen at. This is because a marker will appear the same size as one twice as large, twice as far away — thus the pixel width of the marker in the image will not change. In practice this is not necessarily true as cameras are not perfect optical systems and lighting differences may change appearance. The practical experimentation, presented in Figure 6.12, show that one very evident trend is the increasing tail off in terms of range for larger markers. This is likely an artifact of the granularity of the testing and, under ideal circumstances where markers could be tested at much smaller intervals, it is probable that all marker sizes would exhibit this curve, just compressed on the x-axis. In addition to this the operable range of the markers increases with their size as expected, with the exception of the 200×200 mm marker. In this case the marker's detection rate deteriorated rapidly after about 5000 mm. This is due to the Kinect's depth sensor which is not reliable beyond this depth, causing the corner extractor to fail — frequently rejecting correct marker detections as it erroneously calculated them to be too large/small. This is a limitation of the system implemented and could be fixed by disabling the corner extractor's check on size for markers where any of the points are over 5000 mm away. This would allow detection of markers but also increase the number of false positives in this range. Extrapolating the trend of previous markers, one might expect that, following this correction, the 200×200 mm marker could be detected from 8000 mm away. For the purposes of this system, however, such large markers are not needed so no marker will be detected in this range, rendering the fix unnecessary. The effect of marker size on range is summarised in Table 6.7 and compared against other solutions.

Marker Size	**Range (mm)**		
	Used Method [1]	**ARToolKit** [44]	**ARTag** [2] [47]
35×35 mm	1000	-	750
50×50 mm	1750	-	1100
75×75 mm	2500	410	1600
100×100 mm	4000	860	2200
200×200 mm	5750	1270	4400

Table 6.7: The results of marker size on the operating range of the marker extractor.

Table 6.7 shows the implemented solution to consistently outperform two of the most widely used approaches to solving the problem of marker detection in terms of detection range. In addition its detection range is unaffected by marker Y-axis rotations up to 45° and is able to continue detecting markers at up to 75°. The ARToolKit [39] and ARTag [42] solutions both have advantages over the implemented method however, namely that both of them have a library of available markers all of which are detected and tracked independently by using combinations of black and white squares in the centre of the tag. ARToolKit provides a tool with which users may make their own tags and is able to simultaneously load descriptions of multiple markers and track them independently. The ARTag library has a built-in library of 2,002 markers which it can track [47] (one of which is shown in 6.3a). By comparison the implemented solution can detect only a single design of marker. Additionally the performance of both

[1] The range was measured as the furthest distance at which at least 40/50 readings could be successfully made.

[2] Results were given as minimum detectable marker size in terms of pixels and converted to the distance at which a marker would have that size when viewed with the Kinect's camera. This may overstate the range as the values given are for a much higher quality (less noisy) camera.

the ARToolKit library (which ran at 76.3 FPS [41]) and the ARTag library (which ran at approximately 50 FPS in comparable conditions [42]) exceed that of the implemented solution (17.2 FPS). Furthermore the implemented solution performs worse at long range than that of Claus and Fitzgibbon [37] who managed 10 m with a 20×20 cm target, however this is due to the Kinect's limited range, without which it is reasonable to believe a range of 8 m could be gained. Claus and Fitzgibbon also built a custom classifier cascade to detect the markers which was not viable for this project due to time constraints. Their solution also performed more slowly (8.3 FPS compared to 17.2). Were additional time invested to this it is likely that considerable range benefits would be observed. The implementation meets the requirements of this project, accurately detecting small markers at medium-range (1–5 m) and the limitation to a single marker design is easily justifiable.

To allow comparison with other work in the field the speed of the implemented solution has been evaluated using a desktop system. Transferring the module to the PandaBoard results in a considerable speed reduction, as would be expected. The system runs continuously at 1.4 FPS, 8% the speed of the desktop. This is a considerable weakness of the PandaBoard, especially since the entire system is serial in nature so such drastic reductions in speed heavily affect the other modules (e.g. the object avoidance module which relies on fast reaction times). Removal of the corner extraction step, leaving just the Viola-Jones classifier, does not improve the speed by a measurable amount. Possible solutions to the problem are running the marker detector on smaller images, for example resizing the camera output from 640×480 pixels to 320×240, however doing so reduces the pixel width of each marker, the same effect as would be observed by moving the markers further away. Halving the number of pixels in each dimension in the input image will consequentially shorten the detectable range by half. Another solution, the one used, is to run the marker detector sporadically rather than every frame. This does not compromise accuracy and, providing the robot does not move excessively quickly, will not miss any markers. Without the marker detector the system runs at 30 FPS (the fastest data may be read from the Kinect) using 50% of one CPU, and running the marker detector only every 20 frames (as was chosen) still allows the marker detector to be run approximately once a second, giving ample opportunity to detect markers. The robot must be stationary when the marker detection system is run in order to avoid uneven temporal gaps between readings in the frame buffer used by the object avoidance module (Chapter 4). Such uneven gaps would cause objects to appear to move far faster than they actually are (since all frames are assumed to be evenly spaced and one would be several hundred milliseconds later than 'expected') — resulting in false positive object detections. Additionally it would prevent the robot reacting quickly to events in the environment since the it is unable to process sensor input while the marker detection is running.

Chapter 7

Occupancy Detection

7.1 Design

The occupancy detection module aims to detect whether a table presented to it currently has one or more people sitting at it. It can be assumed that it will only be run when there is a table in front of it. There is a variety of potential solutions to this problem, a few of which are discussed in Section 2.3.6. Both Klomark [33] and Marín-Hernández and Devy [34] approached the problem of examining whether a single seat in a car was occupied or not. In both cases they used depth information from a stereo vision setup and calculated occupancy based on whether or not enough points fell within a given region of 3D space. Arras *et al.* [35] used a laser scanner to collect depth readings of an office setting and applied a machine learning approach to identify people's legs.

One solution to the problem which would produce a robust and general purpose occupant detector (having the fewest requirements in terms of positioning and environment to successfully detect occupants) is to make use of the Viola-Jones object detector, as used for marker detection in Chapter 6 and described in Section 2.2.7. The exact approach used by Chapter 6, applying it the colour video stream, is not necessarily applicable in this instance, however. The Viola-Jones detector works by detecting similar features in terms of contrast in the image, for example in the context of face recognition, it can detect the contrast between darker regions around the eyes and the lighter regions of the cheeks. For leg recognition, however, there are no such features which are consistent across all legs: their appearance depends entirely on the material of the clothing worn by the individual and the background (which may be either lighter or darker than the leg in question). This hypothesis was proved by carrying out a short experiment. In this an object detector was trained on a relatively small training set of 500 positive and 500 negative samples (the 500 positive samples were made up of 100 unique, hand labelled images of legs, and a further 400 images automatically generated from them, produced with the same method described in Section 6.2). While this is a fairly small training set — far too small to generate accurate classifications — it would allow any potential in the idea to be shown. The resulting detector, when applied to a video stream, failed to detect any legs and generated vast numbers of incorrect classifications, suggesting, as expected, that the method is unable to determine such poorly defined features such as legs in colour images. The Viola-Jones method would be better suited to detecting legs in a grayscale version of the depth image (where lighter objects are further from the camera) since, in that case, the background would always be lighter than the object, giving a fixed, sharp-edged contrast which could be exploited by the detector. Unfortunately in order to test this theory a large training set would be needed, something far more difficult to generate for depth images than colour since there are no published libraries of such images. Additionally complicating matters is the fact that depth images are much more difficult to automatically generate because, unlike colour images which may have cropped elements inserted to create believable results, there is meaning construed in the gradients and contrasting edges in a depth image (namely 3D structure). Were part of a depth image to be cropped into another it would leave a wholly unnatural contrast around the edge which could be wrongly interpreted by the object detector's training process. Thus several thousand positive and negative images would have to be taken and labelled manually, a task not within the scope of this project due to time constraints.

Another possible solution would be a machine learning approach, similar to that employed successfully by Arras *et al.* [35]. This has the advantage of being a general purpose solution, capable of detecting legs in different environments, and data similar to that used by Arras *et al.* could be produced with the Kinect, however it would be considerably more noisy than that from their laser scanner (which, as discussed in Section 2.4, was a high-end model). The increase in noise makes it unlikely that a solution as accurate as they produced could be realised.

The approach of Klomark [33] and Marín-Hernández and Devy [34] can be considered roughly equivalent to histogram analysis, as employed in Chapter 4, in that the approaches both look for a given number of depth readings within a certain range to indicate the presence of a person. While this does not produce a general solution — there is no guarantee the readings in a given range relate to the object that is being searched for — in the case of this project it is acceptable since the distance from the table at which the robot will stop is known (as described in Section 6.1). Basic histogram analysis would also not be robust to noise, the presence of things such as bags and other objects under the table would produce false positive readings, however Marín-Hernández and Devy [34] faced a similar issue in their system with child seats which when unoccupied, could wrongly appear occupied. They solved the problem by splitting the 3D space into regions and considering each of these regions separately. A similar solution is applicable to this project: by considering only regions of the image off the floor and unlikely to contain anything other than legs it is possible to greatly reduce the false positive rate. Manual selection of this region is possible from a collection of sample images.

7.2 Implementation

The chosen solution, as discussed in Section 7.1, was to use histogram analysis to determine whether there were a significant number of depth readings in a given range from the camera when positioned 800 mm from the marker, directly facing it (which is expected to be installed on a corner of the table). It is necessary for the robot to be at least 500 mm away from any legs encountered due to the Kinect's minimum range (described in Section 2.2.1) and further distance from the marker reduces the likelihood of objects being placed within this range.

In order to reduce the number of false positives generated from other objects in the scene, such as seats and items that might be left on the floor, only a small region of the image was used to build up the histogram. As the distance and orientation the robot will stop from the table is known, this region is sufficient to remove background 'clutter' from the depth image, and extract only depth readings from directly under the table. This is far from a foolproof method however, and false readings will be generated if the robot stops, for example, facing a wall rather than a table, causing it to wrongly mark the wall on the map as a table.

For this project it is not necessary to know the exact number of legs in the image, rather just whether or not there is at least one. If it were required to know the exact number of legs a system similar to that used by Arras *et al.* [35] and Greuter *et al.* [27] could be implemented whereby the depth data from the region is projected down the Y-axis to produce a 'bird's-eye-view' of the scene. The data may then be clustered (to extract points that are near to each other and therefore are assumed to make up an object) and the similarity in shape of each cluster to a circle computed. Clusters with a shape sufficiently similar to a circle may then be assumed to be legs (as in the case of Arras, or 'game elements' in the case of Greuter). This method is not used by this project, however, as it is more computationally expensive than just histogram analysis, which can impart the information needed.

The region was picked manually based on a variety of sample images taken of a variety of occupied and unoccupied tables. This region, shown in Figure 7.1, uses $1/3^{rd}$ of the data in the image. The corresponding histogram produced from that region is shown in Figure 7.2. It is worth noting that one of the legs has been missed in the case of Figure 7.1 as it has fallen outside of the marked region, however the users other leg falls within the region.

Having identified the region to analyse it is necessary to select the range in the corresponding histogram to compare. As it is known that the viewpoint will be approximately 800 mm from the table the start of the range can be reasonably assumed to be 900 mm. This will ignore the table leg on which the

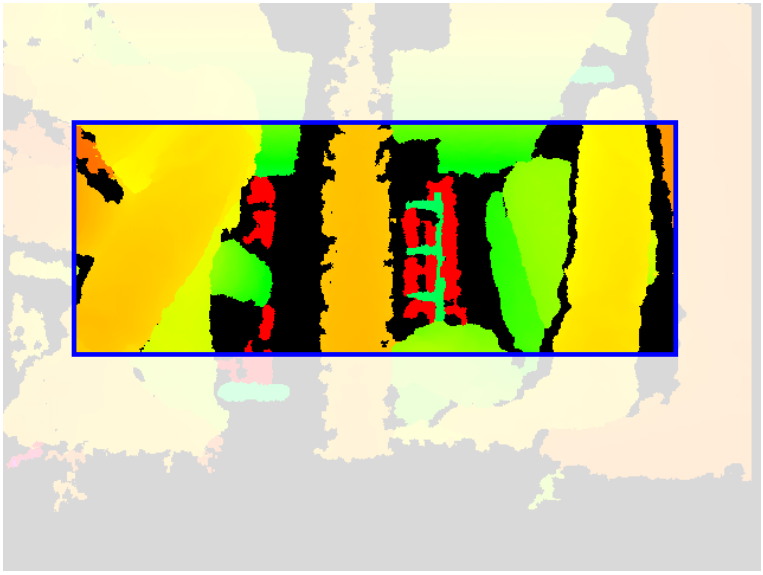


Figure 7.1: The region, shown within the blue border, used to extract the histogram for occupant detection.

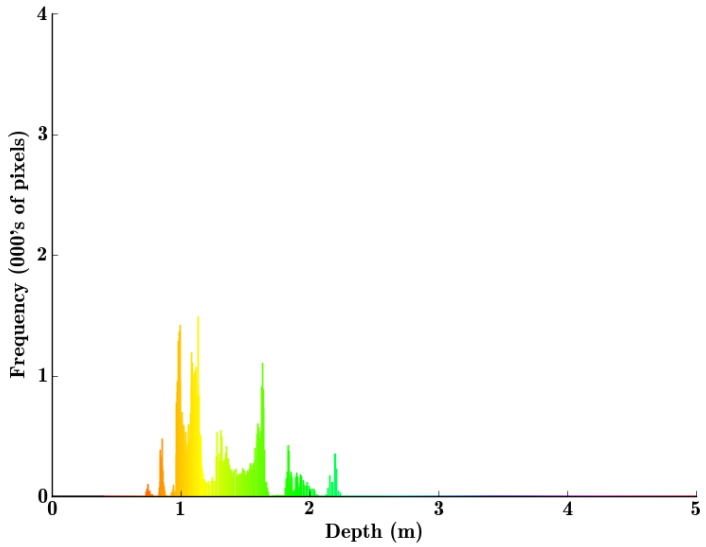


Figure 7.2: The histogram produced from the region in Figure 7.1.

marker is placed and allow the robot to be 100 mm further away than expected, a reasonable margin for error. Calculating the end of the range may be done approximately by measuring the depth of the table, a common value for which is 1200 mm, and subtracting the 100 mm inset from the marker. This gives a rough range of 900–1900 mm.

The exact value for the end of the range was then calculated, along with the threshold to use, by gathering a collection of sample images. These were made up of 16 depth images of different types of tables, 8 with occupants and 8 without. Use of such a small dataset is sub-optimal, however, as described in Section 7.1, there are no published libraries of such images so all data has to be collected manually — a time-consuming task. These images consisted of a table and a single person sitting in profile to the camera (thus resulting in only a single leg being visible). This offered the least difference between positive and negative sample images and it was assumed that if a table with one person sitting in profile could be correctly classified, a table with more people (and thus more depth readings in the relevant range) could also be. Subsequent testing of the result on images with multiple people (e.g. Figure 7.1) proved this to be the case. The sample images included tables with and without walls behind.

Having collected the sample images the sum over the relevant range in each histogram was calculated. Table 7.1 presents this data as the minimum, maximum and average sum over the range listed for the sample images.

	Number of pixel readings in the given range (mm)			
Image set	**900–1700**	**900–1800**	**900–1900**	**900–2000**
Negative (min–max)	978–10,358	978–10,516	978–11,112	8,069–28,142
Positive (min–max)	12,056–29,107	12,056–29,111	12,056–32,086	18,370–42,728
Negative (average)	5,003	5,304	6,214	11,797
Positive (average)	20,275	21,237	23,778	29,618

Table 7.1: The sum of pixel frequencies in varying ranges of the test data collected.

Table 7.1 shows that it is possible to fully split the data with 100% accuracy using a threshold of 12,000 pixels in any of the ranges 900–1700, 900–1800 or 900–1900 mm. The range 900–2000 mm shows a marked increase in the number of readings in both positive and negative images suggesting the inadvertent inclusion of some item of the scenery. Ultimately the range 900–1900 mm was chosen as it gives the greatest difference between the average number of depth readings in that range for the positive and negative images.

7.3 Evaluation

7.3.1 Testing

As discussed in Section 7.1 the occupant detector splits the sample set of depth images gathered with a 100% accuracy. This can be considered its best-case performance, when the camera is lined up with the table correctly, looking directly down its length. However for practical usage it is necessary to investigate the detector's robustness, how resilient it is to sub-optimal positioning. In order to do this the robot was placed facing a table at an angle and the number of pixels extracted from the region's histogram in the 900–1900 mm range noted. A positive classification is made when this value rises above 12,000 pixels, as determined in Section 7.1. The results of this test are presented in Table 7.2.

	Number of pixel readings	
Camera angle	**No occupants**	**One occupant**
0°	7,961	23,551
15°	8,059	30,144
30°	10,186	17,411
45°	12,118	26,295
60°	21,693	25,977
75°	41,993	36,523

Table 7.2: The number of pixel readings generated in the required range when the robot's angle to the table is altered.

Additionally the occupant detector's sensitivity to accurate distance from the marker was tested using the same setup as used for the previous test, in this case adjusting the robot's distance from the table. The results to this test are shown in Table 7.3.

Finally the occupant detector was tested on the scene shown in Figure 7.1 which has three occupants. The number of pixel readings from this image is 65,941 — well in excess of the threshold.

Table distance (mm)	Number of pixel readings	
	No occupants	One occupant
700	17,777	33,048
750	10,541	25,667
800	7,961	23,551
850	7,306	26,044
900	14,359	33,174

Table 7.3: The number of pixel readings generated in the required range when the robot's distance from the table is altered.

7.3.2 Analysis

Table 7.2 shows that the occupant detector is robust to changes in angle up to 30°, after which it starts to generate false positives. This is to be expected as the region being searched contained a wall behind the table which, with increasing angle, moved more and more into the range being searched. This is especially a problem for tables with bench-like seating rather than conventional chairs since this will have a solid base.

Similarly Table 7.3 shows that the occupant detector is robust to distance miscalculations up to $\pm$50 mm, after which it starts to generate false positives. Again this result is unsurprising, if the robot is too close to the table its considered range will include the 3D region behind the table which, in this case, contained a shelving unit. Likewise, being too far from the table causes objects in front it to be considered, including the table leg on which the marker is mounted.

It is worth noting that in the tests carried out a threshold of 17,000 rather than 12,000 would have improved performance, increasing the maximum operable angle to 45° and the maximum operable distance from the table to 900 mm without generating any false negatives. A threshold of 17,000 would still allow a 94% accuracy on the sample data. Furthermore in none of the experiments was a false negative generated.

This method can therefore produce highly accurate results at detecting occupancy on a table which is moderately robust to poor positioning, well within what is achievable by the system, however still prone to inaccuracy if the camera is not pointed correctly under a table. While the solution is limited to working at a distance of 800 mm from the table is is possible for it to move directly forward having identified occupancy to reduce the distance between itself and any occupants at the table, allowing them to dispose of rubbish more conveniently.

Chapter 8

System Evaluation

Figure 8.1: The final robot, complete with its outer shell. The white Roomba can be seen at the base and the Kinect is just visible above that. The PandaBoard and all electronics are hidden from view, mounted in the space behind the Kinect.

8.1 Testing

8.1.1 Functional Testing

Comparing the modules to their specification (Section 3.2) reveals each has met the majority of its goals.

The path planning module meets all of the criteria laid out for it, successfully producing maps of the environment which it uses to revisit areas, preferring to explore when first exposed to the environment and transitioning to exploit the discovered resources over time. Its inability to adjust to external influence limits its success in highly crowded areas, however in areas where it may be easily observed and avoided this is not a problem. Furthermore, while it is possible for it to load an old map it must

be placed in the same starting position and angle each time. This could be an easily identified point however, for example its charging station.

The target recognition module meets all of its goals: it is a medium-range marker detector capable of detecting without loss of range at angles up to 45°, easily sufficient for the project's needs. Anecdotal evidence suggests it is sufficiently robust to changes in lighting to be able to reliably detect markers underneath tables, although the acceptable limits of this were never formally evaluated. Having detected a marker its shape is then extracted and the Kinect's depth data utilised to determine its position. The orientation of the marker is only as accurate as this depth data, so large areas of noise on the marker reduce the module's ability to correctly determine orientation, although an approximate answer is still given. Again, for the purposes of this project any slight inaccuracies in calculating marker orientation or position are within acceptable bounds. The computational expense incurred by the target recognition module is a weakness, however.

Similarly the occupancy detection module is able to accurately determine whether there are occupants at a table, even when there is just one person sitting in profile. It is moderately robust to poor positioning, however its accuracy does fall off at angles greater than 45° or distance error greater than ±50 mm.

The object avoidance module, on the other hand, is one of the weaker parts of the system. It succeeds in that the robot does not drive into obstacles and is capable of identifying even small objects such as chair legs (vital for this project), however it generates a large number of false positive results, frequently causing the robot to wait or turn unnecessarily, wasting time and limiting the exploration of the environment. This is partly caused by the Kinect's data being highly noisy, exacerbated by poor positioning of the Kinect meaning it cannot accurately detect the floor. Furthermore the method does not adequately account for the camera shaking while moving, causing some items at the periphery of the scene to become visible in a single frame and trigger false-positive detections. Additionally the robot sometimes struggles to correctly identify free space after turning that it may move into.

Comparing the resulting system to the original specification (Section 1.3) reveals most of these criteria have also been met. The areas where the robot falls down are its potentially obstructive behaviour — although every decision throughout the project was made to try and avoid this — because the robot moves quite slowly, making it unable to get out of the way of an approaching person in time. Furthermore if it were to block an aisle or corridor this slow movement speed could obstruct people's paths. This is not an easy problem to solve and one of the inherent difficulties involved with creating a robot designed to interact with people in potentially confined spaces. All other goals are met, however.

8.1.2 System Testing

The resulting system meets the high-level goal of creating an autonomous litter bin: the robot does not need external help or input and is capable of recognising, visiting and storing the locations of tables in its internal map. However before it could be deployed successfully in a real world scenario the object avoidance module would need further work to reduce the amount of false positive detections of obstacles, causing it to frequently stop or change direction and making it very slow at navigating the environment as well as potentially obstructing other users.

The PandaBoard and Kinect also perform well together: without the target recognition module the system runs at 30 FPS using 50% of one of the PandaBoard's two cores.

8.1.3 Power Consumption

8.1.3.1 Kinect

An ordinary USB video camera may use a single USB connection to both connect to and draw power from the computer. The USB 2.0 specification allows 5 V ±5% at a maximum of 500 mA to be drawn by a connected device [48]. This is sufficient for the vast majority of USB cameras, however, the Kinect is equipped with a 12 V motor in the base to tilt the device up to 27° up or down. As a result of this it can not draw the required voltage from a USB cable and instead uses a proprietary cable to connect to the PC and draw power. This cable is connected to the Kinect on one end with a USB and 12 V plug on

the other. However, this project does not require the tilt mechanism and so uses significantly less than the Kinect's advertised power consumption of 13 W (12 V at 1.08 A). The actual power consumption was measured using an *efergy* Energy Monitoring Socket (EMS-UK); an in-line power monitor capable of reading power consumptions in the range 0.2–2900 W with an accuracy of ±2% [49]. The results of this are shown in Table 8.1.

Action	Power Consumption (W)	Current (mA)	Voltage (V)
Start Up	2.88	240	12
Idle	1.13	94	12
Capturing	1.51	126	12

Table 8.1: The results of power consumption tests performed on the Kinect.

As illustrated in Table 8.1, the Kinect draws substantially less power for the project's application than is needed for the motor. This is confirmed by a breakdown of the individual components, an analysis of which shows the imaging functionality (the IR projector, IR camera and RGB camera) has a theoretical maximum power consumption of 3.4 W [50]. The main consumer in this case is a cooling device in the IR projector which, during normal use, consumes far less than its theoretical maximum power, hence why the results in Table 8.1 are lower than might be expected.

As such the Kinect is a suitable choice for a low-power system, especially given the additional data it provides. As a comparison a standard USB video camera uses 5 V at up to 500 mA (as provided by the USB connection), equivalent to 2.5 W. The other Kinect-like devices, discussed in Section 2.4, are powered uniquely by the USB, meaning they also consume at most 2.5 W.

8.1.3.2 PandaBoard

The PandaBoard was chosen due to the fact it is low-power, and is powered in this project by an *Anker* Astro3E battery pack which has a capacity of 10000 mAh which can supply 5 V at 3 A. The power consumption of the PandaBoard was measured using an *efergy* Energy Monitoring Socket, as described above. The results of this are shown in Table 8.2.

Action	Power Consumption (W)	Current (mA)	Voltage (V)
Start Up	1.64	328	5
Idle	1.27	254	5
100% CPU load	1.48	296	5
200% CPU load	1.64	328	5

Table 8.2: The results of power consumption tests performed on the PandaBoard.

The results shown in Table 8.2 are for the average power consumption under the stated conditions, however it has been reported [51] that the PandaBoard can draw in excess of 500 mA for short periods of time. As such it cannot be powered over a single USB 2.0 port (which, as mentioned previously, allows 5 V ±5% at a maximum of 500 mA to be drawn by a connected device [48]) — a common output on most battery packs. In this project a Y-cable was used to split the power consumption over two USB ports.

In the case of this project the power consumption need not be as high as it is, however a default Desktop distribution of Linux was used (discussed in Section 2.2.2) which includes a graphical interface as well as a variety of drivers for hardware which is not used (e.g. Bluetooth). If a distribution were custom built for the purpose it is likely additional power savings would be observed.

8.2 Analysis

Such an effective and computationally inexpensive solution would not be possible without the use of the Kinect's depth data, which, as hypothesised, allows a variety of traditionally difficult tasks associated with robotic systems to be done far more efficiently. This is especially true of the histogram analysis technique used in both the object avoidance and occupancy detection modules, which is able to consider the contents of regions of 3D space. In the case of the object avoidance module it was not as successfully deployed as it could have been because the solution was not sufficiently robust to noise. Results from this could be improved by changing the mounting of the Kinect so it was more securely attached to the Roomba, minimising vibrations of the camera. For prototyping purposes it was only attached with Velcro, which led to excessive vibrations causing some false positive detections (discussed in Section 4.3). Additionally an improvement on the same technique considering multiple depth regions and further splitting the image into smaller chunks could see large performance increases and would be necessary before the robot could be viable commercially. The occupancy detection module, however, demonstrated the power of the technique as it allowed a very accurate and simple classifier to be built that split the sample data exactly (discussed in Section 7.3). The threshold used by the technique could be calculated more accurately with the collection of further sample data, improving results.

The target recognition module showed the limitations of the PandaBoard as it was unable to be run every frame, instead running every 20th (discussed in Section 6.3). This meant that the robot had to be stopped during the module's execution to avoid damaging the performance of the object avoidance module, slowing the progress of the robot and preventing it from moving smoothly. Theoretically this allows the robot to miss some markers that it travels past in the 20 frames between target recognition runs, however in practice this is unlikely due to the small distance travelled in that time. Improvements could be made to this solution by creating a custom algorithm to search uniquely for the black and white marker, replacing the general purpose Viola-Jones object detector used. Additionally if a marker were used which was not rotationally symmetric its orientation could be calculated without the use of the Kinect's depth data (using the width and height in pixels of the detected marker), removing the range restrictions imposed by it. For rotationally symmetric markers it is impossible to discern whether the marker is oriented positively (e.g. facing to the left) or negatively (e.g. facing to the right) from just a single camera. More importantly however, the fact that the target recognition module reduced to just 8% of the execution speed when transferred from a desktop computer to the PandaBoard highlights the potential problems associated with using the Kinect and a single-board computer as the sole computer vision system. If the problem cannot be made easier by the presence of the depth data (as it cannot for marker detection) conventional algorithms are hamstrung by the limited processing power available.

The path planning module illustrates that a simple set of rules selected probabilistically, while requiring a negligible amount of computing power to evaluate, may produce sufficiently sophisticated results. In the case of this project the path planning was made significantly easier by the absence of the need to localise within the environment, something which would greatly help the overall reliability of the system. If localisation within the map is required then a SLAM technique must be used. There are many SLAM algorithms which are able to use depth data [52, 53, 54]. Oliver *et al.* [54] in particular achieve a real-time system in conditions similar to those of this project with the Kinect, although using a laptop rather than a single-board computer.

8.3 Alternatives

Other 3D-imaging cameras are available in place of the Kinect. Other 'Kinect-like' devices, those using the same PrimeSense technology as the Kinect (discussed in Section 2.2.1), are theoretically already compatible with the project and could be used interchangeably. This is due to the use of the OpenNI library [6] which abstracts the specifics of the connected camera and provides a generalised interface for it, usable with any supported camera. Additional to Kinect-like devices other 3D-imaging solutions are available, as discussed in Section 2.4. The Kinect was chosen due to its ready availability, off-the-

shelf functionality (requiring no calibration) and extremely low cost (a tenth that of a comparative laser rangefinder).

Alternatives to the PandaBoard are presented in Table 2.1 and its choice discussed in Section 2.2.2. It was chosen mainly as it was the most computationally powerful single-board computer which was readily available. Alternatives to single-board computers were also considered, however while these offered a moderate performance increase they were also considerably larger and had almost five times the power consumption.

The Roomba was chosen as the motorised base for the project because it was readily available and had an easily accessible (if basic) interface through which it could be controlled. It is also circular, an advantage for this project as discussed in Section 2.2.3. The most obvious alternative solution would be to custom build a robotic system for the project, however this was not feasible due to time constraints.

8.4 Other Applications

The autonomous litter bin is designed primarily for large spaces which generate a lot of litter and have a high throughput of customers, for example retail food courts. In this scenario the bin will roams the area visiting tables to collect small amounts of litter. It would then visit a drop-off point after a certain number of table visits to deposit the collected litter (although this functionality is not currently implemented). The bin could also be sold as a high-value piece of furniture. The success of the Sony Aibo, a robotic dog which retailed for around $1600, shows there is a market for high-value robotics.

The system also has a number of alternative applications besides collecting litter. It can be easily applied for use in large buildings such as hospitals or care homes for transporting laundry. It would be ideally suited to this task if it were given a map, which would be feasible if it were to be deployed in a single large complex, as it could travel between wards collecting laundry and transporting it to a drop-off point. This saves a human worker doing the job and would work well in the long corridors associated with such places.

Similarly the system could be deployed in an assistive robotics application for use by elderly people. In this instance it would be required to follow them around their home, transporting items for them that were otherwise too heavy or difficult to carry. This could also be extended for use in the garden, helping with the transportation of gardening tools.

Additional modifications to the robot's structure would allow the system to be used as a robotic waiter to service tables. Having learnt the environment the path planning system would allow the robot to locate the correct table and deliver the food.

The combination of the Kinect and PandaBoard comprise a low-power computer vision system which is perfectly suited for a wide range of robotics applications. As discussed in Section 8.2 it is not necessarily a suitable system for applications which cannot be realised more efficiently with the Kinect's added depth data, however the majority of problems would benefit to some degree from it. Compared to the alternative, a standard laptop with a built in webcam, the system is lighter (1.5 kg compared to an average laptop weighing 3 kg), cheaper (£250 compared to £400 for an average laptop) and has a longer battery life (22 hours (using the values in Tables 8.1 and 8.2) compared to 3 hours for a standard laptop). These are all attractive properties for a computer vision system in robotics applications and it could be successfully deployed for a vast range of robotics projects.

Chapter 9

Conclusions

9.1 Achievements

This project successfully demonstrates the feasibility of a low-power computer vision solution using the PandaBoard and Kinect. This is not a combination which has been tried frequently before — as discussed in Section 2.3.2 only one other academic project was found coupling the technologies. The project would not have been possible without the Kinect, confirming the original hypothesis that its addition allows traditionally complex computer vision algorithms to be performed more efficiently, so much so that they may be executed on a single-board computer resulting in savings in terms of power consumption. This was tested practically by implementing an autonomous litter bin using the Kinect and PandaBoard system. This was able to function as required and, although the object avoidance module in particular requires further work to produce robust behaviour, it serves to prove that the system is viable. The limitations of the system were also exposed by the target recognition module which was unable to run at significantly over 1 FPS due to its computational complexity.

The histogram analysis technique, used by the object avoidance and occupancy analysis modules, was shown to be particularly powerful considering its low computational cost.

Additionally, in implementing the target recognition module a marker detector was built which was able to detect and extract the shape of markers with an operable range in excess of that of commonly used systems for such tasks. While this has the downside that it can only detect a single design of marker where other systems may have large libraries of markers, this is not a limitation in the context of this project. Only one marker detector was able to perform at a greater range and it ran more slowly and used a custom built object detector, where this project used an existing implementation of the Viola-Jones detector. With the development of a custom detector it is believed that the system would be capable of extracting markers at even greater distances.

9.2 Summary of Work Undertaken

In this project the following things have been implemented from scratch:

- The DBSCAN algorithm — as described by Ester *et al.* [3], written out in Section A.1 and used by Section 4.1.2.
- Dijkstra's algorithm — as described by Dijkstra [4], written out in Section A.2 and used by Section 5.1.3.
- The Box-Muller transform — as described by Box and Muller [5], written out in Section A.3 and used by Section 5.1.1.
- Connected-Component Labelling — described in and used by Section 6.1.

- The Frame Buffer — described in and used by Section 4.1 (implemented as a circular buffer, described in Section 2.2.5).
- The marker extraction procedure — described in and used by Section 6.1.
- The histogram analysis procedure — described in and used by Section 4.1 and to an extent Section 7.1.
- The marker pattern design — described in and used by Section 6.1.
- The path planning procedure — described in and used by Section 5.1.
- The library of functions for controlling the Roomba via its "Open Interface " (a simple serial protocol).

The following existing implementations have been used:

- The Viola-Jones object detector — described in Section 2.2.7 and used by Section 5.1. This implementation is from the OpenCV library [2].
- The viewpoint transformation — described in and used by Section 6.2. This implementation is from the OpenNI library [6].
- The JPEG Compression — described in and used by Section 4.2. This implementation is from the OpenCV library [2].
- Image thresholding — described in and used by Section 6.1. This implementation is from the OpenCV library [2].
- The tool for automatically generating training images — described in and used by Section 6.2. This implementation is from the OpenCV library [2].
- General framework provided by the OpenCV library [2], for example the `Mat` class for storing matrices and operating with them.
- General framework provided by the OpenNI library [6], for example methods for configuring and extracting data from the Kinect.

Additionally the hardware used, specifically the Kinect, PandaBoard and Roomba, are all existing technologies described in Sections 2.2.1, 2.2.2 and 2.2.3 respectively.

9.3 Evaluation

The autonomous bin was able to meet all of its goals to some degree, as discussed in Section 8.2. The object avoidance module, while functional, could be improved to reduce the number of false positive object detections which both slow the system down and cause erratic behaviour. A number of methods by which the module could be improved are presented in Section 4.3.2, however none have been implemented due to time constraints. The target recognition module, while functional and sufficiently accurate, could also be improved to reduce the time it takes to run. This could be done by building a custom object detector rather than using an off-the-shelf implementation (as detailed in Section 6.3.2), however this is a considerable and time consuming task, not within the scope of this project. The path planning and occupant detection systems, on the other hand, succeed in meeting all of their objectives.

The system as a whole also meets all of its objectives (as laid out in Section 1.3), although it is more obstructive in its behaviour than intended. All decisions made in the project have focused around reducing the likelihood of the robot being obstructive, however due to the imperfect nature of the object avoidance module and its tendency to generate false positive readings the robot moves more slowly and stops more frequently than would be ideal. Improvement in this module would reduce obstructive behaviour greatly.

The combination of the PandaBoard and Kinect to produce a low-power computer vision system has also been successful, as demonstrated by its deployment in the autonomous bin. It is possible to combine these technologies to create a system which is lighter, cheaper and has a longer battery life than the alternative laptop-based solution.

9.4 Project Status

9.4.1 Open Problems

The main weakness of the current system is the object avoidance module. Given more time this could be greatly improved and the technique refined, as described in Section 4.3.2. Additionally the occupancy analysis module would benefit from a greater number of samples on which to base its threshold for classification.

One of the weaknesses of the approach used is that the robot is unable to see objects approaching from behind. If a rear-facing camera were added it could detect when it was blocking people's paths and ammend this by moving out of the way. The camera would also help speed up changing path because the robot's front would be identical to its back, allowing it to simply change which camera it was using for detecting oncoming objects and reverse its motors — avoiding the need to rotate 180°. Another weakness is the current approach used for path planning which cannot recover if the robot is acted on externally in any way. This is a sizeable weakness, especially considering the robot is designed to operate in highly dynamic environments, however calculating current position while simultaneously constructing a map of the environment is a difficult (but not insurmountable) problem. Section 8.2 presents a number of possible solutions for this.

9.4.2 Extensions

One obvious extension to the project is to expand the classification of 'litter generation' to more than just people sitting at tables, for example targeting groups of people standing up. The bin could also track the trajectory of thrown items and move to catch them if necessary. A further extension would be to to exploit the PandaBoard's dual-core design by implementing a parallel framework, something which was not done during the project due to time constraints and is thus not taken advantage of. Such a framework would be especially useful in the case of the target recognition module which, rather than requiring the robot to stop so it may search for possible markers, could run continuously on the second core. This would leave the primary core free to execute all other modules, namely the object avoidance, path planning and occupant detection. Even greater improvements in speed may be observed if the PandaBoard's GPU were utilised to perform low-level calculations, particularly in the context of the Viola-Jones algorithm which would be able to exploit the GPU's ability to quickly generate and evaluate the integral images it uses (described in Section 2.2.7).

As discussed in Section 8.1.3, modifying the system's Linux distribution from the default Desktop one used could improve its performance and decrease the power consumption by removing and disabling unneeded components. Likewise experimenting with the other Kinect-like cameras would likely allow further power savings to be realised since they are powered uniquely over USB so have a lower maximum theoretical power consumption (2.5 W compared with the Microsoft Kinect's 3.4 W, as described in Section 8.1.3).

9.4.3 Further Work

Additional work on the marker extractor built for the target recognition module could produce a system capable of detecting markers at distances in excess of the operable ranges of any known system. The module could be improved, as described in Section 6.3.2, by constructing a custom detection framework similar to that used by the Viola-Jones object detector, but specialised for searching for a black and white marker — similar to the approach taken by Claus and Fitzgibbon [37] for their marker

detector. If a marker is used which is not rotationally symmetric its orientation may be calculated without the need for the Kinect, removing its limitation on the detector's maximum range and generalising the solution.

Bibliography

[1] azt.tm, "Kinect pattern uncovered," April 2011, available online at http://azttm.wordpress.com/2011/04/03/kinect-pattern-uncovered/, accessed 17th April 2013.

[2] "OpenCV," Willow Garage and Itseez, 2013, available online at http://opencv.org/, accessed 2nd May 2013.

[3] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*. AAAI Press, 1996, pp. 226–231.

[4] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[5] G. E. Box and M. E. Muller, "A note on the generation of random normal deviates," *The Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610–611, 1958.

[6] "OpenNI | the standard framework for 3D sensing," Open Natural Interaction, 2013, available online at http://www.openni.org/, accessed 2nd May 2013.

[7] J. Leonard and H. Durrant-Whyte, "Simultaneous map building and localization for an autonomous mobile robot," in *Intelligent Robots and Systems '91. Intelligence for Mechanical Systems, Proceedings IROS '91. IEEE/RSJ International Workshop on*, vol. 3, 1991, pp. 1442–1447.

[8] M. Watanabe and S. K. Nayar, "Rational filters for passive depth from defocus," *International Journal of Computer Vision*, vol. 27, no. 3, pp. 203–225, 1998.

[9] B. Freedman, A. Shpunt, and Y. Arieli, "Distance-varying illumination and imaging techniques for depth mapping," US Patent US 2010/0 290 698 A1, November 18th, 2010.

[10] K. Khoshelham and S. O. Elberink, "Accuracy and resolution of Kinect depth data for indoor mapping applications," *Sensors*, vol. 12, no. 2, pp. 1437–1454, 2012.

[11] R. El-Laithy, J. Huang, and M. Yeh, "Study on the use of Microsoft Kinect for robotics applications," in *Position Location and Navigation Symposium (PLANS), 2012 IEEE/ION*, 2012, pp. 1280–1288.

[12] Business Wire, "PrimeSense teams up with ASUS to bring intuitive PC entertainment to the living room with WAVI Xtion," January 2011, available online at http://www.businesswire.com/news/home/20110103005276/en/PrimeSense-Teams-ASUS-Bring-Intuitive-PC-Entertainment, accessed 18th April 2013.

[13] OMAPpedia, "PandaBoard FAQ, is TI subsidizing the PandaBoard?" October 2012, available online at http://omappedia.org/wiki/PandaBoard_FAQ#Is_TI_subsidizing_the_PandaBoard.3F, accessed 17th April 2013.

[14] iRobot Corporation, *iRobot Create Open Interface (OI) Specification*, 2006, available online at http://www.irobot.com/filelibrary/pdfs/hrd/create/create%20open%20interface_v2.pdf, accessed 17th April 2013.

[15] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1, 2001, pp. I–511–I–518.

[16] R. Lienhart, A. Kuranov, and V. Pisarevsky, "Empirical analysis of detection cascades of boosted classifiers for rapid object detection," in *Pattern Recognition*, ser. Lecture Notes in Computer Science, B. Michaelis and G. Krell, Eds. Springer Berlin Heidelberg, 2003, vol. 2781, pp. 297–304.

[17] R. Lienhart and J. Maydt, "An extended set of haar-like features for rapid object detection," in *Image Processing, 2002. Proceedings. 2002 International Conference on*, vol. 1, 2002, pp. I–900–I–903.

[18] B. Mazzolai, P. Salvini, and P. Dario, "Urban service robotics: Challenges and opportunities," in *15th International Conference on Advanced Robotics (ICAR)*, vol. 15th, June 2011, available online at http://www.icar2011.org/files/Mazzolai_ICAR2011-WS%20Urban%20Robotics.pdf, accessed 17th April 2013.

[19] J. Stowers, M. Hayes, and A. Bainbridge-Smith, "Altitude control of a quadrotor helicopter using depth map from Microsoft Kinect sensor," in *Mechatronics (ICM), 2011 IEEE International Conference on*, 2011, pp. 358–362.

[20] M. T. Draelos, "The Kinect up close: Modifications for short-range depth imaging," Master's thesis, North Carolina State University, Raleigh, North Carolina, 2012.

[21] M. Kepski and B. Kwolek, "Fall detection on embedded platform using Kinect and wireless accelerometer," in *Computers Helping People with Special Needs*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7383, pp. 407–414.

[22] R. Barnett, "TiroKart Blog, category Kinect," 2012, previously available online at http://tirokartblog.wordpress.com/category/kinect/, last successfully accessed 22nd October 2012.

[23] ——, "TiroKart: the total immersion remotely operated go-kart," 2012, available online at http://pandaboard.org/content/tirokart-total-immersion-remotely-operated-go-kart, accessed 2nd May 2013.

[24] R. Barnett and S. Ellis, "SyntroKinect," Pansenti, 2012, available online at http://www.pansenti.com/wordpress/?page_id=1706, accessed 17th April 2013.

[25] R. Mojtahedzadeh, "Robot obstacle avoidance using the Kinect," Master's thesis, KTH School of Computer Science and Communication, Stockholm, Sweden, 2011.

[26] The CogX Project, "Dora the Explorer," available online at http://cogx.eu/results/dora/, accessed 18th April 2013.

[27] M. Greuter, M. Rosenfelder, M. Blaich, and O. Bittel, "Obstacle and game element detection with the 3D-sensor Kinect," in *Research and Education in Robotics — EUROBOT 2011*, ser. Communications in Computer and Information Science, D. Obdržálek and A. Gottscheber, Eds. Springer Berlin Heidelberg, 2011, vol. 161, pp. 130–143.

[28] D. Fox, W. Burgard, F. Dellaert, and S. Thrun, "Monte carlo localization: Efficient position estimation for mobile robots," in *Proceedings of the National Conference on Artificial Intelligence*. JOHN WILEY & SONS LTD, 1999, pp. 343–349.

[29] J. Bruce and M. Veloso, "Real-time randomized path planning for robot navigation," in *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, vol. 3, 2002, pp. 2383–2388.

[30] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Computer Science Dept, Iowa State University, Tech. Rep. 98-11, 1998.

[31] P. Benavidez and M. Jamshidi, "Mobile robot navigation and target tracking system," in *System of Systems Engineering (SoSE), 2011 6th International Conference on*, 2011, pp. 299–304.

[32] R. Jain, R. Kasturi, and B. G. Schunck, *Machine Vision*, 5th ed. McGraw-Hill, 1995.

[33] M. Klomark, "Occupant detection using computer vision," Master's thesis, Linköping University, Linköping, Sweden, 2000.

[34] A. Marín-Hernández and M. Devy, "Application of a stereovision sensor for the occupant detection and classification in a car cockpit," in *Proceedings of 2nd International Symposium on Robotics and Automation*, 2000, pp. 491–496.

[35] K. Arras, O. Mozos, and W. Burgard, "Using boosted features for the detection of people in 2D range data," in *Robotics and Automation, 2007 IEEE International Conference on*, 2007, pp. 3402–3407.

[36] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[37] D. Claus and A. W. Fitzgibbon, "Reliable fiducial detection in natural scenes," in *Computer Vision — ECCV 2004*, ser. Lecture Notes in Computer Science, T. Pajdla and J. Matas, Eds. Springer Berlin Heidelberg, 2004, vol. 3024, pp. 469–480.

[38] L. Belussi and N. S. T. Hirata, "Fast QR code detection in arbitrarily acquired images," in *Graphics, Patterns and Images (Sibgrapi), 2011 24th SIBGRAPI Conference on*, 2011, pp. 281–288.

[39] H. Katoa, M. Billinghurstb, and I. Poupyrev, *ARToolKit User Manual, version 2.33*, 2nd ed., Human Interface Technology Lab, University of Washington, 2000.

[40] H. Katoa, "ARToolKit," Human Interface Technology Lab, University of Washington, 2013, available online at http://www.hitl.washington.edu/artoolkit/, accessed 6th May 2013.

[41] X. Zhang, S. Fronz, and N. Navab, "Visual marker detection and decoding in AR systems: A comparative study," in *Proceedings of the 1st International Symposium on Mixed and Augmented Reality*, ser. ISMAR '02. IEEE Computer Society, 2002, pp. 97–106.

[42] M. Fiala, "Artag, a fiducial marker system using digital techniques," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 2, 2005, pp. 590–596.

[43] P. M. Wayne, W. Piekarski, and B. H. Thomas, "Measuring ARToolKit accuracy in long distance tracking experiments," in *In 1st International Augmented Reality Toolkit Workshop*, 2002.

[44] U. o. W. Human Interface Technology Lab, "How does ARToolKit work?" available online at http://www.hitl.washington.edu/artoolkit/documentation/userarwork.htm, accessed 8th May 2013.

[45] M. Fiala, "ARTag," 2009, available online at http://www.artag.net/, accessed 6th May 2013.

[46] J. Wu, S. C. Brubaker, M. D. Mullin, and J. M. Rehg, "Fast asymmetric learning for cascade face detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 30, no. 3, pp. 369–382, 2008.

[47] M. Fiala, "ARTag revision 1, a fiducial marker system using digital techniques," Computational Video Group, Institute for Information Technology, National Research Council Canada, Tech. Rep., November 2004.

[48] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips, *Universal Serial Bus Specification*, April 2000.

[49] efergy, *Energy Monitoring Socket 2.0 Data Sheet*, 2011, available online at http://www.efergy.com/media/download/datasheets/ems_uk_datasheet_web2011.pdf, accessed 15th April 2013.

[50] The OpenKinect Project, "Kinect hardware information," February 2011, available online at http://openkinect.org/wiki/Hardware_info, accessed 16th April 2013.

[51] OMAPpedia, "PandaBoard FAQ, can PandaBoard be powered through USB?" (sic.), October 2012, available online at http://omappedia.org/wiki/PandaBoard_FAQ#Can_PandaBoard_be_powered_through_USB.3F, accessed 8th May 2013.

[52] N. Engelhard, F. Endres, J. Hess, J. Sturm, and W. Burgard, "Real-time 3D visual SLAM with a hand-held RGB-D camera," in *Proc. of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum, Vasteras, Sweden*, vol. 2011, 2011.

[53] A. P. Gee and W. Mayol-Cuevas, "6D relocalisation for RGBD cameras using synthetic view regression," in *Proceedings of the British Machine Vision Conference (BMVC).(September 2012)*, 2012.

[54] A. Oliver, S. Kang, B. C. Wünsche, and B. MacDonald, "Using the Kinect as a navigation sensor for mobile robotics," in *Proceedings of the 27th Conference on Image and Vision Computing New Zealand*, ser. IVCNZ '12. New York, NY, USA: ACM, 2012, pp. 509–514.

Appendix A

Additional Algorithms

A.1 DBSCAN

The DBSCAN algorithm is used to group a dataset into clusters according to their spatial density.

Algorithm 4: The DBSCAN Algorithm [3] (density-based spatial clustering of applications with noise).

```
Input  : D − The dataset.
Input  : ε − The locality of points.
Input  : m − The minimum number of points needed to form a cluster.
Output: C − The set of clusters.
C ← 0
foreach unvisited p ∈ D do
    mark p as visited
    N ← all points less than ε distance from p
    if |N| < m then
        mark p as noise
    end
    else
        C ← next cluster
        add p to C
        foreach p′ ∈ N do
            if p′ is unvisited then
                mark p′ as visited
                N′ ← all points less than ε distance from p′
                if |N′| ≥ m then
                    N ← N ∪ N′
                end
            end
            if p′ is not yet in a cluster then
                add p′ to C
            end
        end
    end
end
```

A.2 Dijkstra's Algorithm

Dijkstra's algorithm is used to calculate the shortest path between two paths in a graph.

Algorithm 5: Dijkstra's Algorithm [4].

Input : G — The graph.
Input : S — The start node.
Input : T — The target node.
Output: R — The shortest path of nodes between S and T in G (inclusive), or $\emptyset$ if no such path exists.

```
for all vertices v ∈ G do
    d_v ← ∞
    p_v ← ∞
end
d_S ← 0
Q ← the set of all nodes in G
while |Q| ≠ 0 do
    u ← vertex in Q with smallest value in d
    Q ← Q \ u
    if d_u = ∞ or 5 then
        break
    else if u = T then
        break
    end
    foreach neighbour v of u do
        a ← d_u + DistanceBetween(u, v)
        if a < d_v then
            d_v ← a
            p_v ← u
        end
    end
end
R ← ∅
u ← T
while p_u ≠ ∞ do
    R ← u ∪ R
    u ← p_u
end
if |R| > 0 then
    R ← u ∪ R
end
```

A.3 Box-Muller Transform

The Box-Muller transform is used to generate normally distributed random numbers using a uniformly distributed source of random numbers. A pseudo-random normal distribution can be created from a pseudo-random source of uniform numbers, for example C's `rand()` function.

Algorithm 6: The Box-Muller Transform [5].

Input : μ — The mean of the distribution.
Input : σ — The standard deviation of the distribution.
Output: Z — A normally distributed pseudo-random number.
$U_1 \sim \mathcal{U}(0,\ 1)$
$U_2 \sim \mathcal{U}(0,\ 1)$
$Z \leftarrow \sigma\big(\sqrt{-2\ln U_1}\cos(2\pi U_2)\big) + \mu$

A.4 Closest Point Behind Current Position

Algorithm 7 describes an algorithm for extracting the closest point r behind a point p when it has orientation γ.

Algorithm 7: Calculates the closest node to a point at a given orientation.

Input : G — The graph.
Input : p — The point to calculate the closest point behind.
Input : γ — The orientation of p.
Output: r — The closest point behind p with orientation γ.
$R \leftarrow \emptyset$
for *all points* $q \in G$ **do**
  $s \leftarrow \big((q_y - p_y)\sin(\gamma)\big) - \big((q_x - p_x)\cos(\gamma)\big)$
  if $s < 0$ **then**
    $R \leftarrow R \cup q$
  end
end
$r \leftarrow$ the point s which minimises DistanceBetween$(p, s),\ \forall s \in R$

Appendix B

Additional Calculations

B.1 Time to Impact Calculations

As illustrated in Figure B.1, knowing the object position p_1 at time t_1 and its position p_2 at time t_2 it is possible to calculate if the object will collide with a circle around the camera at radius r and, if it will, the time of impact t_i and its potion p_i.

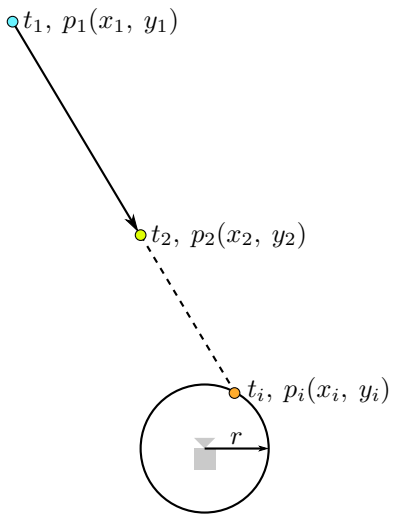


Figure B.1: A diagram showing the calculating of the impact time and position from two known object positions at different times.

Defining:

$$\begin{aligned} d_x &= x_2 - x_1 \\ d_y &= y_2 - y_1 \\ d_h &= \sqrt{d_x^2 + d_y^2} \end{aligned} \tag{B.1}$$

and calculating the determinant:

$$D = x_1 y_2 - x_2 y_1 \tag{B.2}$$

allows calculation of the discriminant:

$$\Delta = r^2 d_h^2 - D^2 \tag{B.3}$$

Knowing this it is then possible to discern whether or not the object will collide with the surrounding circle as follows:

$$\text{Collision} = \begin{cases} 1 & \text{if } \Delta \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{B.4}$$

If a collision would occur the point of impact $p_i(x_i,\ y_i)$ can then be calculated using the following:

$$x_i = \frac{Dd_y \pm \frac{d_y}{\text{abs}(d_y)} d_x \sqrt{\Delta}}{d_h^2} \tag{B.5}$$

$$y_i = \frac{-Dd_x \pm \text{abs}(d_y)\sqrt{\Delta}}{d_h^2} \tag{B.6}$$

where the function abs(x) calculates the non-negative value of x. This gives two values for x_i and y_i, representing the two points at which the line will pass through the circle (once entering and once leaving). As it is only necessary to know the point at which the line enters the circle values of x_i and y_i should be selected that minimise the distance to p_2. The time of impact t_i is then given by:

$$t_i = t_2 + \frac{|p_i - p_2| \cdot (t_2 - t_1)}{|p_2 - p_1|} \tag{B.7}$$

This example holds for a stationary viewpoint, however it can be easily adjusted for moving viewpoints provided they are moving forwards at a constant speed (as it would be in the context of this project). To do this it is necessary to adjust the value of p_2 as follows:

$$p_2 = \big(x_2,\ y_2 - m(t_2 - t_1)\big) \tag{B.8}$$

where the viewpoint is moving forwards at m units per second.

Appendix C

Additional Graphs

C.1 Greedy Node Action Probability

Figure C.1 shows how the probability of a Greedy Node action being performed increases over time. Values shown assume: updates are performed on average every 15 seconds; that table weights are increased by 1 every 6 seconds and that there are 2 discovered tables (both having a start weight of zero at time zero). It shows Pr(Greedy Node) (calculated using Equation 5.1) — the probability that, if updated the Greedy Node action would be selected, as well as Pr(Greedy Node), Cumulative — the probability that a Greedy Node action would have been performed by that point in time. This illustrates that while Pr(Greedy Node) may be comparatively low, the fact that it is sampled multiple times means that as time increases it becomes more and more likely that at least one sample produces a Greedy Node action (at which point one of the table weights would be reset to zero).

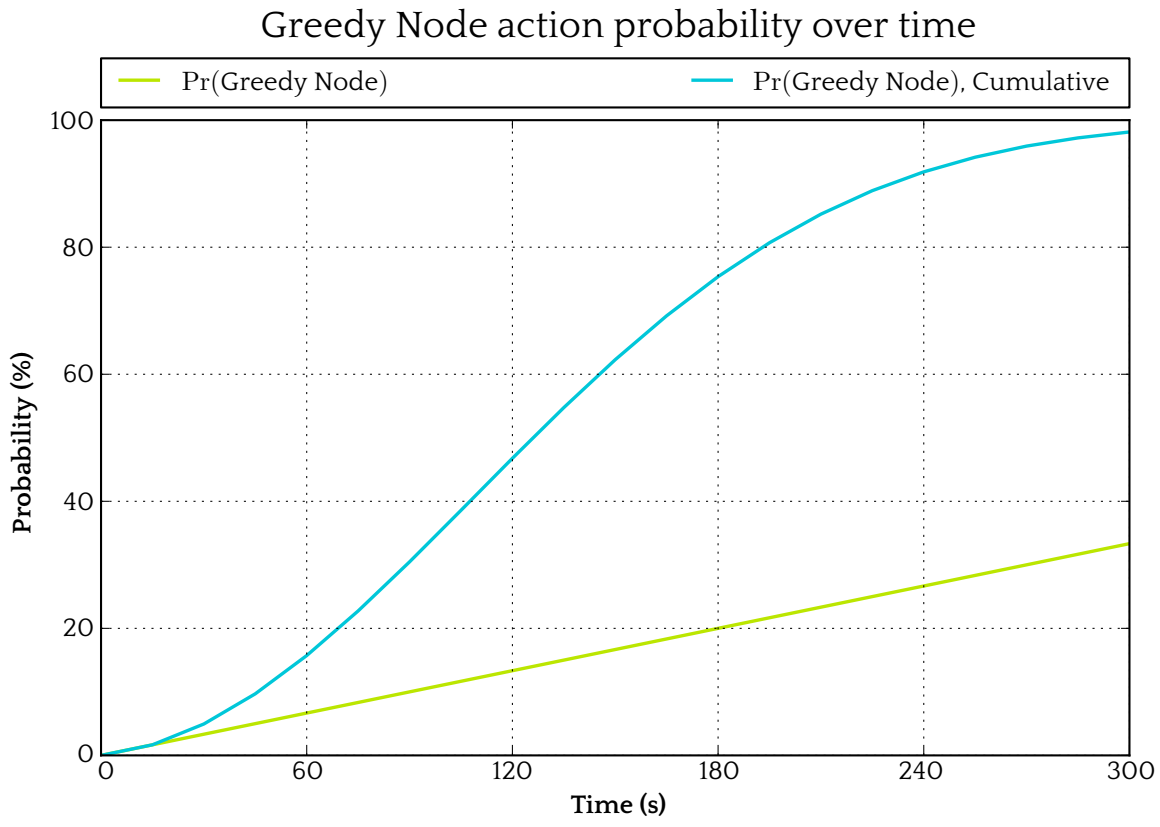


Figure C.1: Graph showing how the probability of a Greedy Node action (Pr(Greedy Node) as defined in Equation 5.1) changes over time.

Appendix D

Additional Figures

D.1 Marker Training Data

Figure D.1: An example of an image automatically generated as positive training data for the object detector.

D.2 Decision Tree Splits

Figure D.2 shows the structure of a decision tree classifier changing as the number of splits (or levels) used by the tree changes. The Viola-Jones object detector was tested with 1, 2 and 3 splits. A 3-split tree is not illustrated.

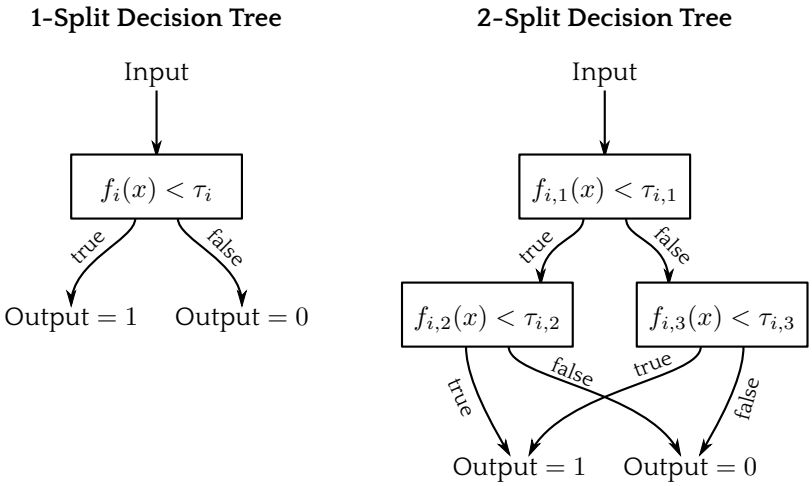


Figure D.2: Diagram showing how the structure of a decision tree classifier changes for one and two splits.