Contents

List of Figures

Chapter 1

Introduction

The word 'navigation' might seem closely related to modern technology for most people, but actually mankind's urge for navigation dates back to the earliest of ages, when navigation was as simple as following the stars in the sky. As we grew, our urge for new navigation techniques grew more and more, and as we evolved our techniques evolved with us. Thus paving the way for mankind's transition into a new age; the age of exploration.

People finally got to understand how their world looks like, and with time, the map of our earth was complete. Then it was time for technology to revolutionize everything. Suddenly, sailing changed from navigating with a compass and a piece of flotsam, to electrical impulses and radio-waves guided navigators. Moving into the modern era, where technology has blossomed, Doppler and Inertial navigation systems became the norm for todays navigation [1]. In addition to the space navigation that allowed us to travel to moons and planets.

Although our reach has gone so far, we are still challenged by areas that are physically impossible for a human being to explore. Areas like accidents in chemical plants, fire fighting, radioactive plants, etc.. Nowadays, researchers are developing autonomous machines that can self-navigate through those unreachable areas, and many researches and projects ([2], [3], [4], [5], [6]) are tacking place in that field. But one typical area is becoming increasingly more common, that is the research in the field of Unmanned Aerial Vehicles (UAVs).

Flight unit for Autonomous Location and Territory Exploration (FALTER) is a research project that is currently being developed by the Software and Systems Engineering research division at Fortiss GmbH. With the mission of providing an effective and efficient support for the exploration of indoor terrain and situations, FALTER aims for developing autonomous UAVs capable of navigating through dangerous unknown environments, performing on-site analysis and returning back to the base-station, where the collected data could be observed and analyzed.

1.1 Aim of Research

In order for FALTER to fly to goal, it needs the ability to plan a safe journey through the unknown environment that lies ahead. Planning a navigation strategy for such journey is a subject of significant interest, due to the broad range of techniques that could be practically implemented and utilized.

1.1.1 Research Objectives and Contribution

The objective of this research study is to present the technique of using range sensing cameras for path planning and navigation in complex 3D spaces, and to prove its validity and effectiveness for such application. Most mobile robot implementations until today rely on 2D sensors, and the usage of depth cameras is not common among researchers. This research contributes in showcasing the potential in the field of 3D sensing and depth vision, as an efficient solution to the problem of environment exploration and navigation. It also provides the basis for further development and innovation in various research directions

1.1.2 Thesis Statment

Due to the possibility of processing their captured data to extract and locate obstacles in unknown environments, depth sensing cameras are considered as an effective, efficient and attractive technique to navigate through complex 3D spaces.

1.2 Scope of Work and Limitations

Throughout this research we will target the ideas behind object detection, path planning and obstacle avoidance for unmanned aerial vehicles. We focus on the use of range finding sensors, mainly kinect depth sensor, for putting those ideas into action. During the development of our software, we were limited to the vision capabilities of the kinect depth sensor. We were also limited by the fact that the kinect can not be integrated with the current hardware setup of FALTER quadrocopter. Such limitation prevented us from applying our software to the copter control unit, for real-life testing.

1.3 Thesis Outline

Chapter 2 provides the basic background needed to understand and appreciate the terms and expressions to be discussed in the subsequent chapters.

Chapter 3 briefly overviews the current state of art developments related to our topic, and summarizes many techniques developed to solve the problems of path planning and navigation.

In **chapter 4** explains the current FALTER hardware and software architecture. Stressing on providing a clear overview about the data acquired from kinect, and how it will be used in serving the purpose of our research.

Chapter 5 discusses five different object detection techniques, and ends by concluding which technique would best fit our application.

Chapter 6 presents the technique used for path planning and obstacle avoidance, using depth data from kinect.

Chapter 7 is explaining how the algorithm developed in chapter 6 was integrated with the virtual reality of FALTER on the matlab simulink simulation.

In **chapter 8** we purposed a technique to further extend the algorithm presented in chapter 6.

Finally, **chapter 9** summarizes the whole thesis, and gives some conclusions based on the testing results. It also demonstrates some directions where the future work on FALTER could proceed.

Chapter 2

Background

2.1 Unmanned Aerial Vehicles

Robots that are intended to fly in air without a pilot or crew on board are called Unmanned Aerial Vehicles (UAVs). UAVs can operate remotely (e.g. controlled by a ground control station) or autonomously based on per-programmed flight plans and more dynamic and complex operational systems. UAVs became such an important tool, not only in the military and special operation applications, but also in the civilian ones, such as firefighting and non-military security work. They are often preferred for missions that are unpredictable and dangerous for manned aircrafts to operate within [12].

2.1.1 Quadrocopters

Quadrocopters are a special kind of UAVs that are lifted and propelled by four rotors. With the growth in the field of Unmanned Aerial Vehicles, quadrocopters are particularly becoming more popular again, especially for observational and exploration purposes in outdoor and indoor environments. Also they are used for data collection, or simply as high-tech toys.

The history of quadrocopters date back to 1920s and 1930s, when a number of manned designs first appeared. However, those early prototypes suffered from poor performance, poor stability and required too much pilot work load [13].

More recent designs became more popular in UAV research. These designs use an electronic control system and electronic sensors to stabilize the aircraft. These vehicles can fly indoors as well as outdoors ,because of their small size and agile maneuverability [14] [15].

Among the advantages of quadrocopters, is that their maintenance is relatively simple compared to other UAVs. Also, they use four rotors each with a small diameter, allowing them to possess less kinetic energy during flight. Moreover, they are considered small-scale UAVs, which makes it easy to surround their rotors with a cage or a frame,

and thus making them safer for close interaction.

In order to move or to change orientation, quadrocopters perform four different kinds of motions, as shown in figure 2.1: Rolling, Pitching, Yawing and Vertical Motion.

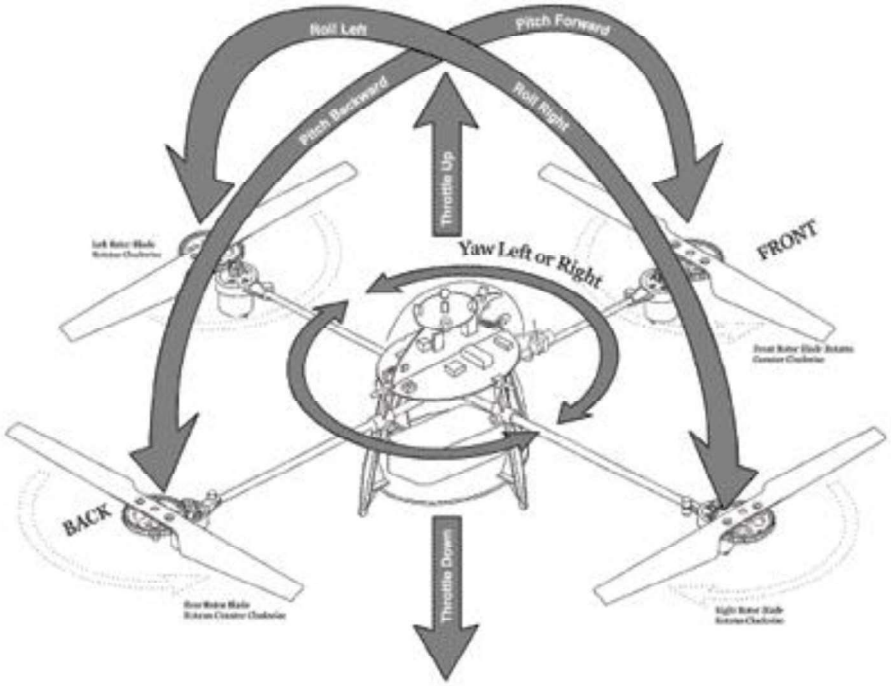

Figure 2.1: Copter Degrees of Freedom

2.2 Computer Vision

Computer vision is the transformation of data from a still or video camera into either a decision or a symbolic representation [20]. This includes methods for acquiring, processing and analysis of data. After performing this transformation into our digital devices, what we record are numerical values for each of the points of the image.

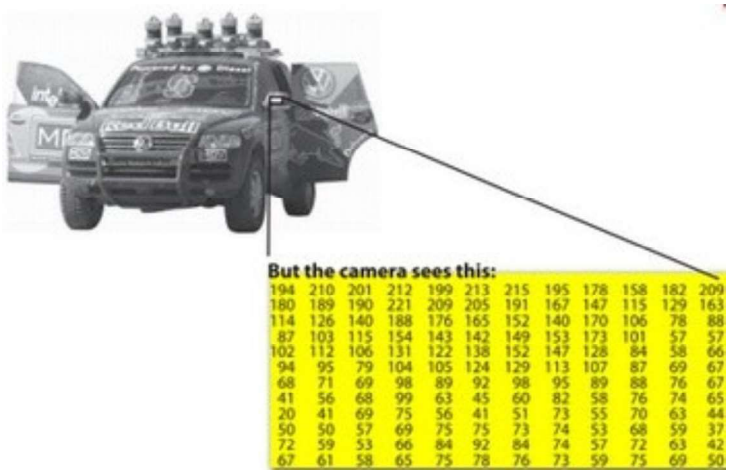

Figure 2.2: Digital image representation

For example, in the above image you can see that the mirror of the car is nothing more than a matrix containing all the intensity values of the pixel points. In the end, all images inside a computer world may be reduced to numerical matrices and other information describing the matrix itself [18].

2.3 Image Processing

Image and vision are widely considered in almost every recent application due to the continuous advances of imaging techniques, and the decrease of computational and financial costs. Therefore, understanding image processing and analysis is fundamental for developing real world systems.

Image processing is a collection of operations and algorithms that are performed on the numerical values of image pixels to enhance or transform the image into new form. The output of image processing may either be an image, a set of characteristics or a set of parameters related to the image.

Digital image processing allows the use of much more complex algorithms, and hence, offering more sophisticated performance at simple tasks, and facilitating the implementation of methods which would be impossible by analog means.

2.4 Image Noise

Since the image is nothing more than a signal, and like all signals, images are prone to noise interferences. In image processing, noise is considered to be random unwanted variations in brightness and color of the image.

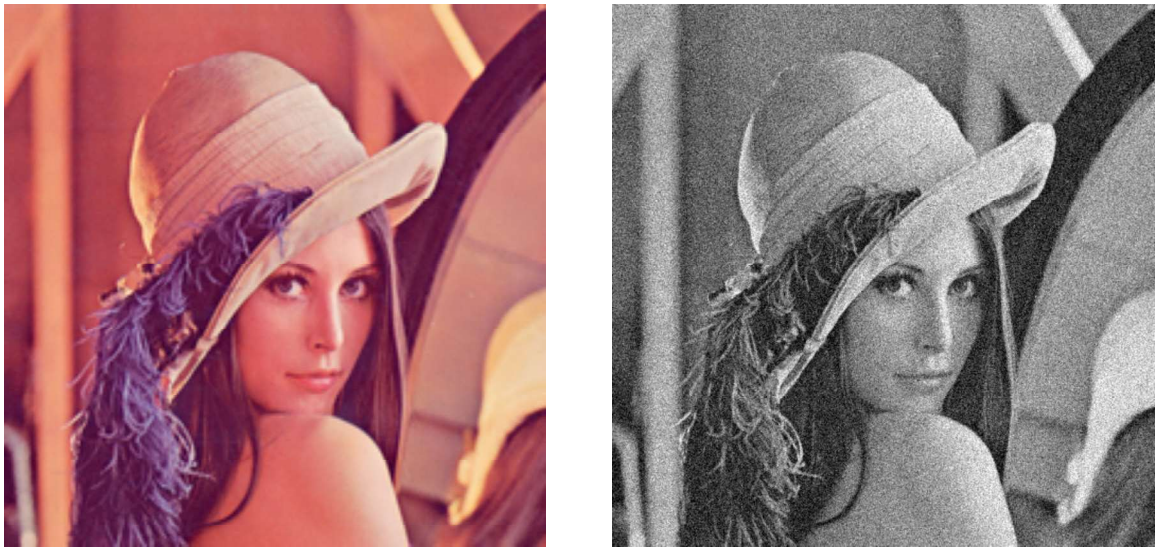

Figure 2.3: Effect of noise on images

2.5 Image Spaces

A digital image is a numeric representation of a 2D image of any visual scene. In this section we discuss some of those representations and how they differ.

2.5.1 Binary Image

Binary images are the simplest form of image representations, where each image pixel accepts only one of two values: '0' representing black and '1' representing white. They acquire very small storage spaces in memory, because they are single channel images where each pixel is only stored in one bit.

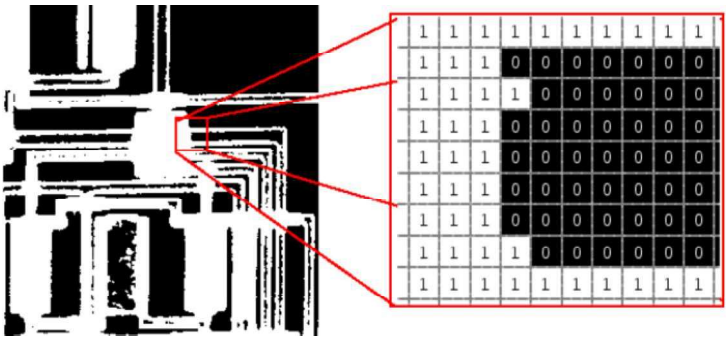

Figure 2.4: Binary image representation

The most common use of binary images is in thresholding during edge detection or object segmentation.

2.5.2 Gray Scale Images

Gray scale images are digital images,where each pixel is carrying information representing its intensity and brightness. The range of intestines lie between $0 - 255$, where '0' represents absolute black and '255' represents pure white, giving us a total of 256 distinct shades of gray in between.

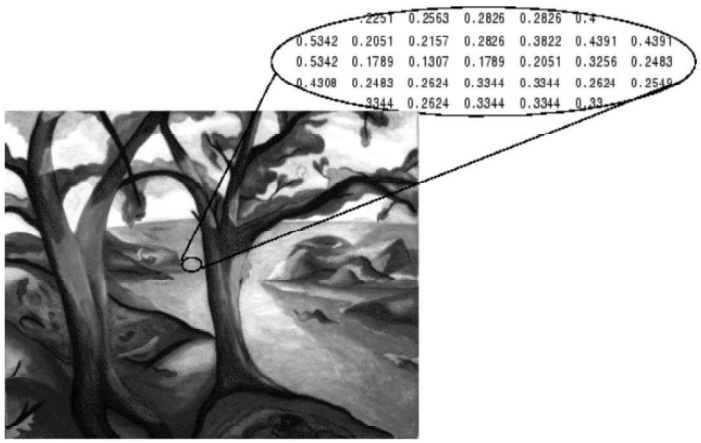

Figure 2.5: Gray scale image representation

There is also a different representation of gray images where intestines range between $0 - 1$. In this case, '0' is black, '1' is white and all decimals values in between represent the different shades of gray color.

Gray scale images are one-channel images, which means that, they are represented in memory by one two-dimensional array of bytes, having the same dimensions as the image width and height.

2.5.3 RGB Images

RGB images, commonly known as colored images, are images formed by mixing different shades of three colors: Red, Green and Blue.

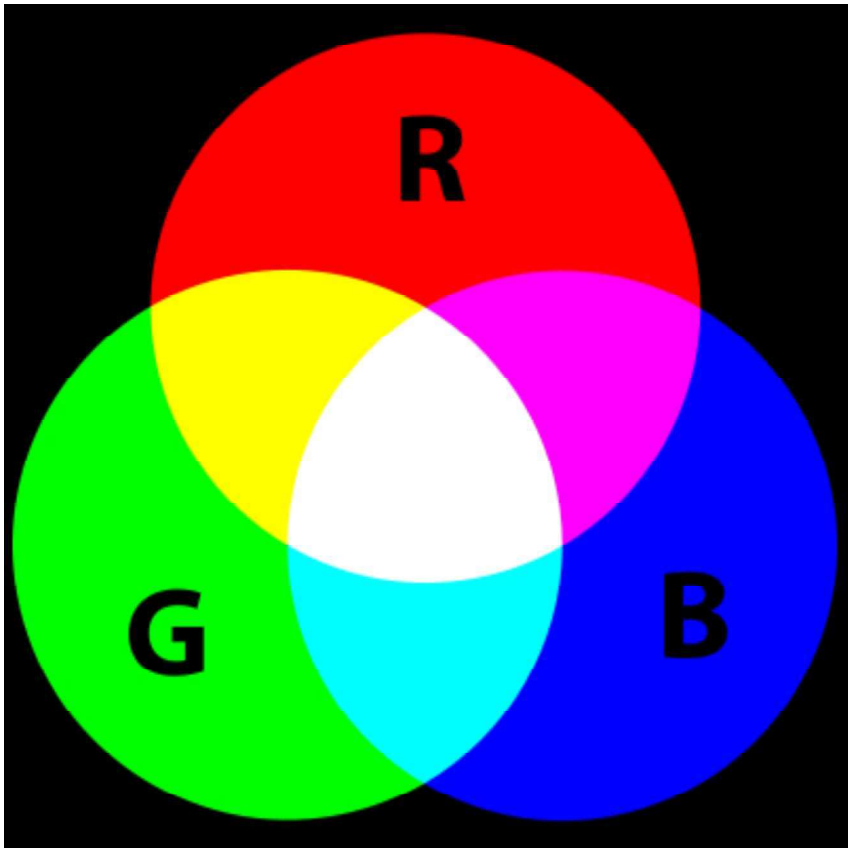

Figure 2.6: RGB Color mixing

Colored images support approximately 16.5 million distinct shades of color. To support this huge number of color options, three bytes of data are dedicated to each pixel. The three bytes are spited between them, so that one byte is dedicated for each color and its 256 shades.

Figure 2.7: RGB image representation

RGB images are three channel images. However, In most computer vision softwares, color images are represented by four channels. This extra channel is called the *Alpha channel*, which represents the transparency of each pixel.

2.5.4 HSV Images

Like RGB, HSV images are three channel images, but with a different representation. HSV color space has a cylindrical geometrical representation (figure 2.8). Hue is the angular dimension, it starts from 0 where the color is red, passing by green at angle 120, blue at 240 and returning back to red at 360. The vertical axis is the intensity (gray level) of each value. It is ranging from dark low intensities at the bottom, getting lighter and lighter moving upwards. Finally, the saturation horizontal axis that represents the amount of color. At the center of the cylinder, where the color is white, saturation in minimum. Then it gets stronger moving towards the edges as the color gets richer and more saturated.

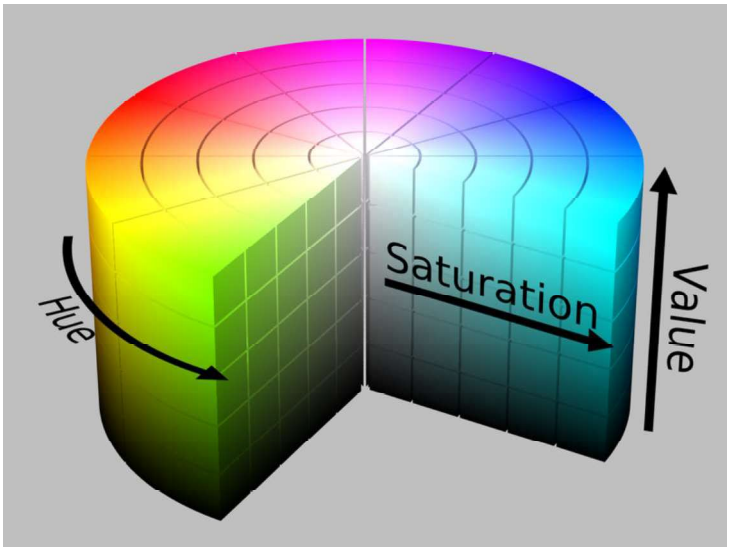

Figure 2.8: HSV image representation

2.5.5 Depth Images

Depth images are images formed by depth sensing cameras. They are very important, because they provide us with an abstract 3D perception of the scene. Depth images are different from normal images, because each pixel in the depth image carries data representing the actual distance between the elements seen by the pixels, and the position of the camera.

Figure 2.9: Depth image

2.6 Depth sensing cameras

Depth sensing cameras are cameras that consist of a matrix composed of distance sensors that deliver two types of information for each pixel: gray value and depth value. In contrast to stereo vision, Depth sensing cameras provide depth values as a direct output, without the need of any further calculations. This sections presents two different types of depth sensing techniques, and there underlying way of operation.

2.6.1 Time of Flight Cameras

Time of Flight (T.o.F) cameras are depth sensing cameras that have the ability to calculate actual distances in a scene based on the known speed of light. They Emmit intensity-modulated light near infra red range. The light signal travels with constant speed in the surrounding medium, and gets reflected by objects in its way.

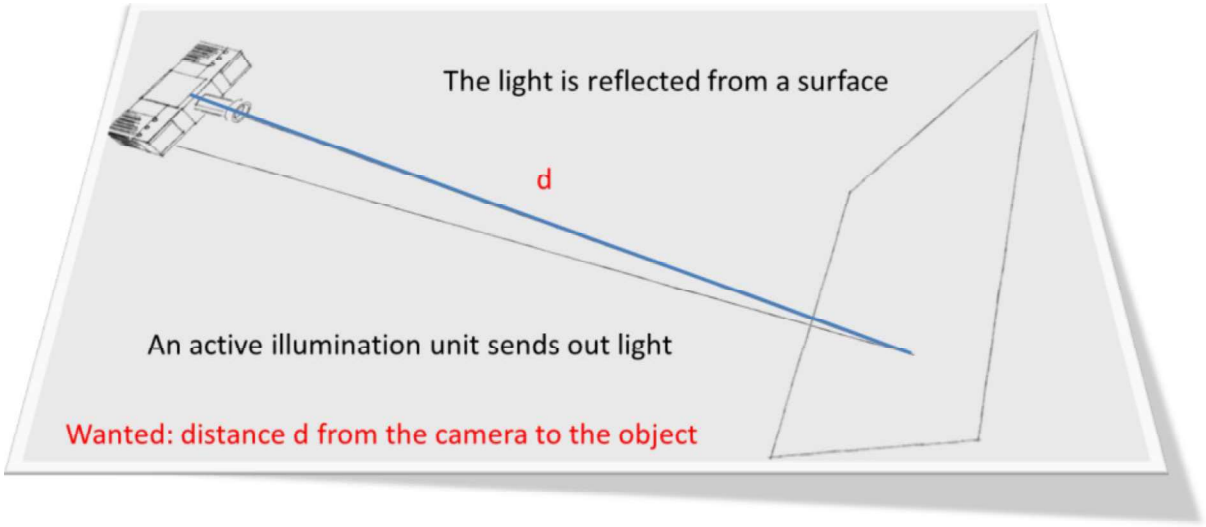

Figure 2.10: ToF camera distance calculation

By estimating the phase-shift between the emitted and reflected light, the distance can be computed as:

$$d = \frac{c}{2f_{mod}} \frac{\phi_d}{2\pi} \tag{2.1}$$

where c [m/s] denotes the speed of light, d [m] the distance the light travels, f_{mod} [MHz] the modulation frequency, ϕ_d [rad] the phase shift.

2.6.2 Structured Light Cameras

The principle of structured light is projecting a known pattern of pixels onto a scene.

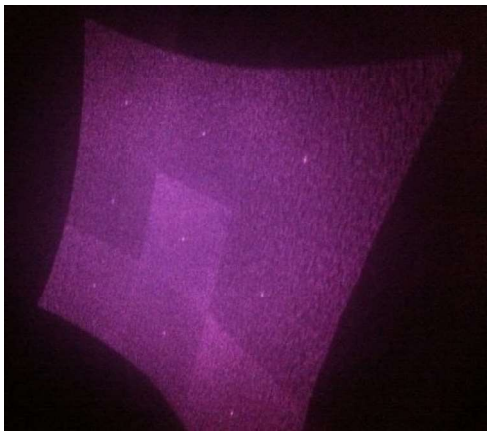

Figure 2.11: structured light pattern visualization

Due to pattern deformation when striking objects in the scene, and given the angle between the sensor and the light pattern emitter, depth could be recovered through simple triangulation.

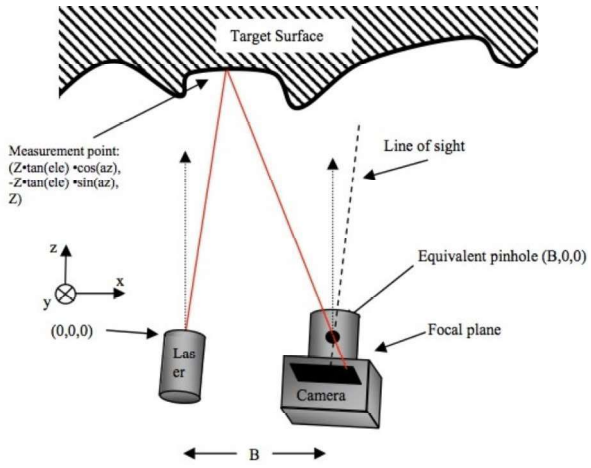

Figure 2.12: structured light distance calculation

The recent application of structured light RGB-D cameras to home entertainment and gaming has resulted in the wide availability of low-cost RGB-D sensors well suited for robotic applications. In particular, the Microsoft Kinect sensor, developed by *PrimeSense* [31].

Chapter 3

State of Art

Throughout the literature and during the past years of research, development in the area of path planning and obstacle avoidance has grown a lot. Nowadays, there are a huge number of algorithms and techniques being put into action. For example, the Bug algorithms presented in [23] [24]. The Bug algorithms assumes only local knowledge of the environment and a global goal. The simple Bug 0 algorithm directs the robot along straight line from start to goal. When it gets to an obstacle, it follows its boundary. While circumventing the obstacle boundary, when the robot does not perceive an obstacle ahead of it along the straight line to the goal, it resumes movement to goal.

A more complicated but less efficient, is the Bug 1 algorithm, where the robot is directed along straight line from start to goal. When it gets to an obstacle, it circumvents obstacle once; calculates a point on obstacle boundary that is closest to goal, heads to that point and resumes movement in straight line towards goal. In the Bug 2 algorithm, the robot calculates straight line from start to goal, denoted as m-line. When it gets to an obstacle, it follows obstacle boundary until it reaches a point on obstacle boundary that intersects with m-line; resumes movement towards goal while following m-line.

A second widely used path planning technique is the Rapidly exploring random tree (RRT). The RRT planner is designed specifically for path planning problems that involve no differential constraints [35]. the RRT constructs a tree that randomly expands in a few directions to quickly explore the map. The technique behind the algorithm is described in details in [26] [44] [16]

Among the most efficient methods today, are the Probabilistic Road Maps (PRMs) [26] [39] [43], the potential field algorithms [33] [34], vector force field histograms [41] [42], Genetic algorithm planners [45], and the evolutionary algorithmic (EA) techniques ([38], [37], [36]).

However, when the dimensionality of the planning problem is low, deterministic algorithms, like graph search algorithms, are usually favored [30]. A number of classical graph search algorithms have been developed for calculating least-cost paths; two popular ones

are Dijkstras algorithm (Dijkstra 1959) and A* (Hart, Nilsson, & Rafael 1968; Nilsson 1980). They both operate the same way, except that A* guides its search towards the most promising states, therefore potentially saving a significant amount of computation. When operating in real world scenarios, the planning graphs are to a great extent inaccurate. To cope with such inaccuracies, extensions of the A* algorithm, like D*, D* Lite and LPA*', have been developed[D*] [8].

Another growing potential technique for obstacle avoidance and path planning, is through the use of stereo cameras([25], [13], [29], [32], [40]). Particularlly, image disparities and visual odometry are being used for feature recognition. Visual odometry refers to the process of estimating a vehicles 3D motion from visual imagery alone. The basic idea is to identify features of interest in each camera frame, estimate depth to each feature (typically using stereo), match features across time frames, and then estimate the rigid body transformation that best aligns the features over time [31].

Last but not least, recent advancements in the field of depth cameras lead to the development of depth vision-based localization and navigation. Such techniques are very critical in the indoor environments, where GPS is no longer reliable. In [27] a kinect depth sensor was used to construct a map of the environment, localize the robot within the map, and plan a path through the map using D* Lite algorithm. A similar approach to the one presented in this thesis has been tackled In [28], where a Swiss Ranger depth sensor was used, and the Vector Force Field Histogram approach was utilized for planning and navigation.

Chapter 4

FALTER Architecture

4.1 Quadrocopter

The basic platform of FALTER quadrocopter is a MikroKopter basic set L3-ME from *HiSystems GmbH*. It comes with frame, motors and all devices that enables it to fly via remote control. The kit also comes with different propellers with a size of 8, 10 and 12 inches. The quadrocopter is powered by four brush-less motors providing a maximum nominal power of 440W. The actual thrust resulting from it, depends on the propeller size [7].

Figure 4.1: FALTER Quadrocopter

Furthermore, this set includes a per-assembled *FlightCtrl* (Version 2.0 ME) that comes with gyroscopes, a 3D accelerometer, an atmospheric pressure meter and preloaded firmware. Their principal is task of keeping the quadrocopter stable In the air[florian]. The brain of the copter which receives all signals and sends all commands is the RB-100 RoBoard. The RoBoard is a single Vortex86DX processor, which is a 32 bit x86 CPU

running at 1000MHz with 256MB RAM. The RoBoard is designed to be used in mobile robots, and therefore has input and output interfaces for SPI, I2C, ADC and USB. To communicate with the RoBoard, the *FlightCtrl* is additionally connected via the serial interface RS232 [8].

For perception of the environment, FALTER is equipped with eight SRF08 ultra sonic range detector modules from *Devantech*, communicating with the RoBoard through I2C protocol. The quadrocopter is also equipped with two infrared range detectors pointing downwards. Since the ultrasonic range detectors provide more accurate results on the short range, and the infrared range detectors provide more accurate information on the long range, FALTER relies only on the readings from the ultrasonic sensors short ranges, and relies only on the IR sensors long ranges [7]. Instead of switching between the two sensors, a different technique to fuse the data from both, has been proposed in [10].

4.2 Kinect

For most people kinect is just a black box bought as an extension to the Xbox 360, and is used for interactive gaming. On the other hand, for people working in the field of research, kinect is a very powerful tool. Kinect itself was first announced on June 1, 2009 at E3 2009 under the code name "Project Natal" [17]

Inside the sensor case, kinect contains the following :

- An RGB camera that stores three channel data.
- An infrared (IR) emitter, that emits infrared light beams.
- Depth camera based on the structured light technology.
- A multi-array microphone, which contains four microphones for capturing sound.
- A 3-axis accelerometer for the rotating base.

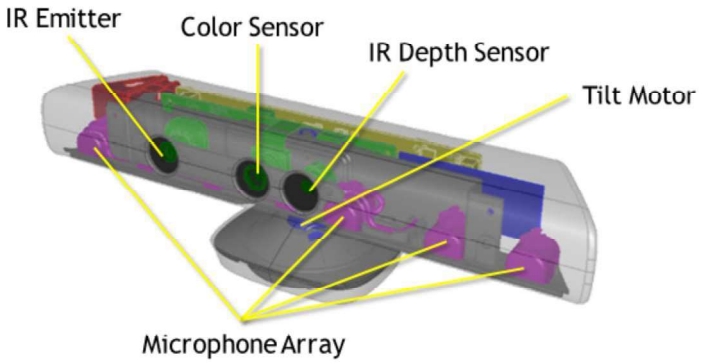

Figure 4.2: Kinect Architecture

Table 4.1: Kinect specifications

Kinect	Array Specifications
Viewing angle	43 vertical by 57 horizontal field of view
Vertical tilt range	27
Frame rate (depth and color stream)	30 frames per second (FPS)
Accelerometer characteristics	A 2G/4G/8G accelerometer configured for the 2G range, with a 1 accuracy upper limit.
Range of Vision	800 4000 mm.

RGB data could be streamed in the following resolutions: $320x240$ or $640x480$ or $1280x960$, and depth data could only be streamed in the following resolutions: $320x240$ or $640x480$. Each resolution affects the number of streamed frames per second. We fixed the resolution to $640x480$ for both cases with a total of 30 fps[msrn].

During processing and simulation, we were streaming RGB and Depth data. However, the kinect can not stream both data in parallel, so we were steaming them consecutively after each other, one RGB frame followed by one depth frame and so on. The total number of frames streamed per second is 30, 15 RGB frames and 15 depth frames.

4.3 Software

The FALTER software and all its algorithms were developed in C++ programming language. To program using kinect, we used Microsoft visual studio 2010 express platform. Furthermore, in the simulation phase we used Matlab 2012, and its simulink extension. In addition, we made use of Matlab s-Function to plug in the C++ code in order to simulate the software in the virtual environment. The software development, testing and the visual simulations were done off-line on a normal desktop computer setting with an Intel core2duo processor, which is a 64-bit CPU running at $3.00GHz$ with $2GB$ of RAM.

4.3.1 Microsoft kinect SDK

Kinect SDK is the Microsoft software development kit for kinect. Its first commercial release was in February 2012. Kinect SDK is a library that allows innovative software engineers and researches to access kinect functionalities. Kinect SDK allowed us to get the kinect working on windows, and to use the data streamed from it.

4.3.2 OpenCV

OpenCV is an open source library providing the tools needed for most computer vision applications. The main focus of openCV is real-time image processing. In our case, openCV high-level functionalities and data types, provided great support and were sufficient for most of our image processing problems.

4.3.3 OpenGL and freeglut

OpenGL is an a library or an interface to the graphics hardware. It is used for drawing 2D and 3D graphics. freeglut is an Open source alternative to the OpenGL Utility Toolkit (GLUT) library. It is written by Pawel W. Olszta with contributions from Andreas Umbach and Steve Baker (http://freeglut.sourceforge.net/). freeglut is used to create and manage windows containing openGL data, and also reads the mouse, keyboard and joystick functions.

In our research openGL and freeglut were used to visualize the output optimal path created by the path planning algorithm in 3D.

4.3.4 RGB Stream

The RGB camera of the kinect returns 4-channel colored images. A memory buffer is created for saving each frame at a time, and they are used throughout the research, especially during object detection and segmentation.

4.3.5 Depth Stream

The depth camera of the kinect returns a depth image containing distance values for each pixel. For safety reasons, we modified the working range of kinect to be within $0.9 - 3.0m$ instead of the default $0.8 - 4.0m$. this is done, by simply overwriting all pixel readings returning values more than $3.0m$ or less than $0.9m$. Finally, the data is feed to the memory buffer hat is especially allocated for it. Figure 4.3, shows how the depth image of the kinect looks like.

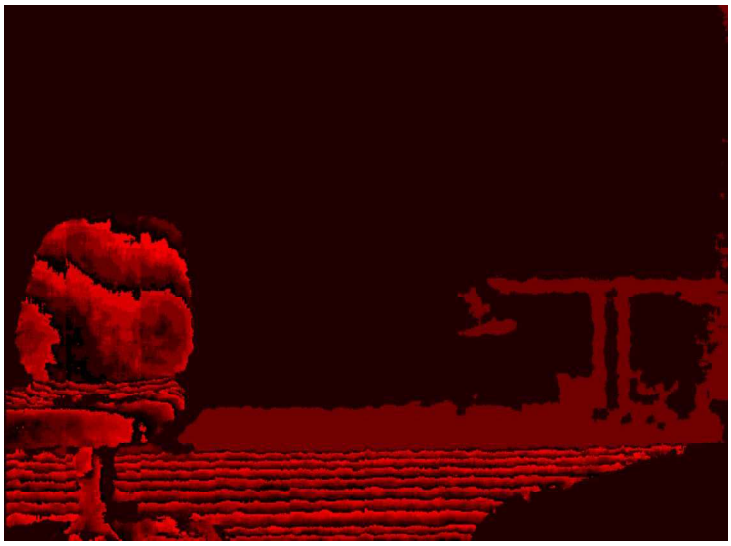

Figure 4.3: Kinect Depth Image

4.3.6 Binary Depth Image

The binary depth image, is a thresholded binary image containing the value '0' for black and '255' for white. All objects within the working range of the kinect appear as white and their corresponding pixels read the value 255. Also, all the objects outside the working range, either closer than $0.9m$ or further than $3.0m$, are suppressed and their corresponding pixels read the value zero. This image is being used for obstacle avoidance and during path planning.

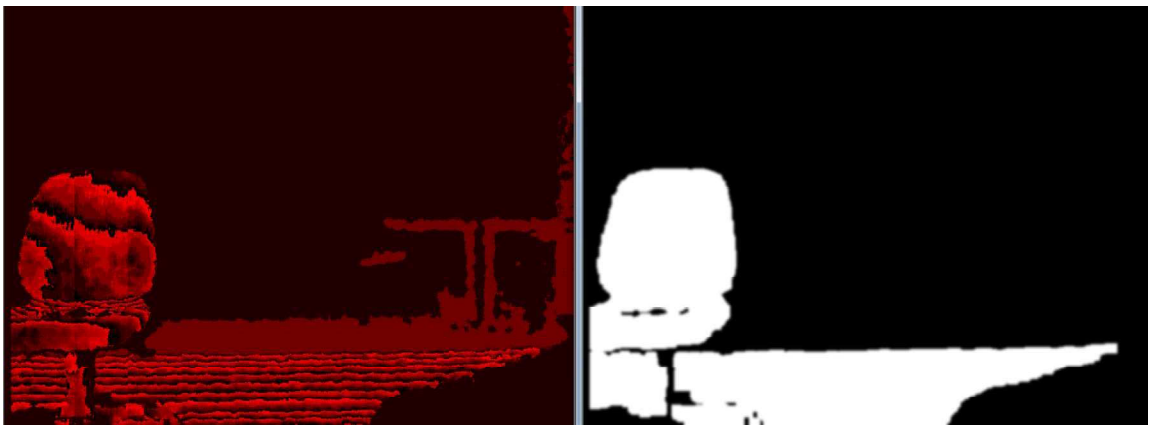

Figure 4.4: Kinect Binary Depth Image

4.3.7 Calculating Real Distances With Respect to Kinect

Thanks to the data from the depth stream, not only do we have an estimate of the depth of the points in the scene, but also we can calculate their real 3D positions with respect to the kinect frame of reference shown below.

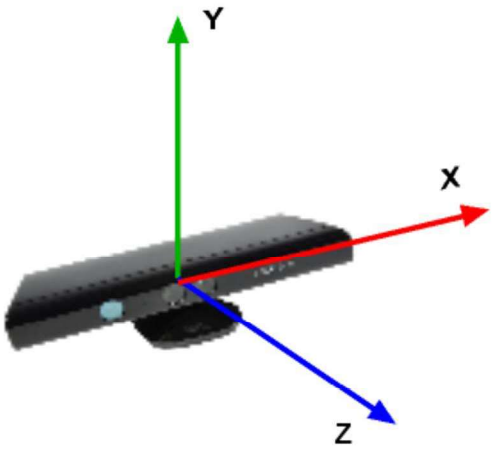

Figure 4.5: Kinect coordinate system

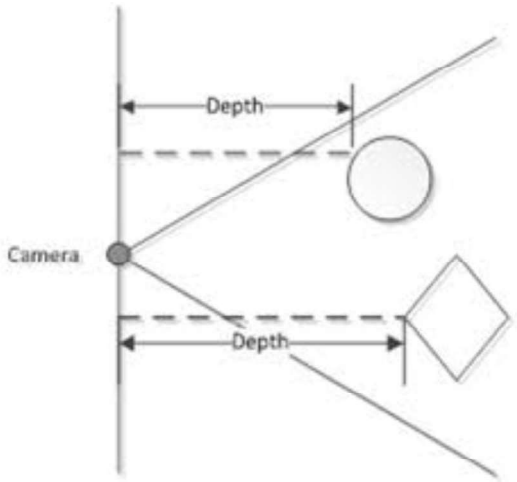

Figure 4.6: Kinect depth mesurment

As shown in figure 4.6, kinect depth sensor always calculates the perpendicular distance to the objects appearing within its field of view. to calculate the X and Y coordinates we need to know how much, in meters, does one pixel represent at a certain depth. With the knowledge of the screen width and height in pixels, and with the knowledge of the horizontal an vertical field of view of the kinect, we can apply the following simple trigonometric equations:

$$pixelwidth[m] = \frac{2dtan(\frac{\theta}{2})}{ImageWidth} \tag{4.1}$$

$$pixelheight[m] = \frac{2dtan(\frac{\phi}{2})}{ImageHeight} \tag{4.2}$$

Where θ [deg] is the Horizontal field of view, ϕ [deg] is the vertical field of view, and d [m] is the depth value.

Finally, since we know the pixel coordinates of the target point with respect to the copter frame of reference, we can calculate the real (X,Y) coordinates by simply multiplying the pixel coordinates by the values obtained from the previous equations.

Chapter 5

Object Detection

For path planning and obstacle avoidance to be put into action, obstacles must first be extracted and detected from a given image or a video stream. Object detection and segmentation is the most important and challenging fundamental task of image processing and computer vision. It is a critical part in many applications including image retrieval and video surveillance. However it is still an open problem due to the complexity of object classes and images.

Throughout this chapter, we will explore some object detection techniques that were implemented and tested on the RGB and/or Depth stream of the kinect. Results were analyzed and the best technique was selected.

5.1 Color detection

Image segmentation by color is the process of extracting, from the image domain, one or more connected regions satisfying uniformity and homogeneity in terms of a chosen color [21].

In color detection HSV color space is used instead of the more common RGB one, where each color degree has a certain unique hue and shading value. This provides a wider range of colors to track and segment. After selecting a certain color to track and after determining its HSV values, comes the last step: thresholding the image, where all image pixels, except for those with the selected HSV values, will be suppressed. Therefore segmenting all objects in the scene having the same color.

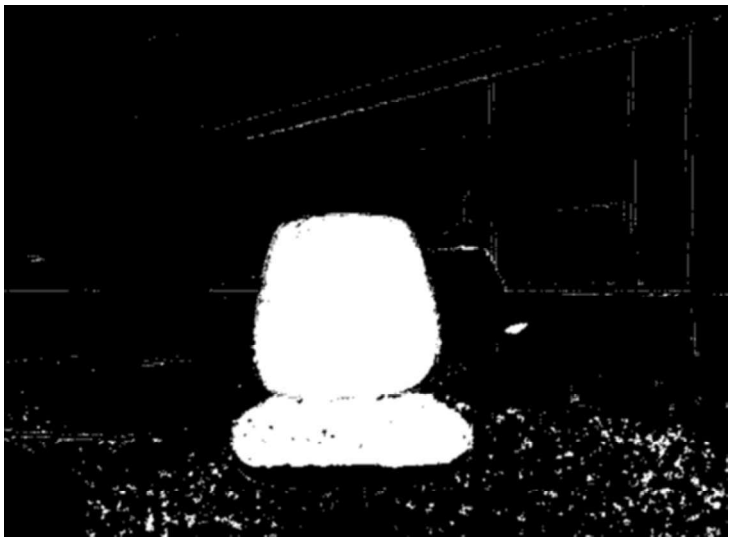

Figure 5.1: Segmentation using color detection

In most cases, one does not exactly know the degree of color one wants to track. As a solution to this problem, we can determine a range of acceptable HSV values to threshold. If a pixel for example, has HSV values between a predefined upper and lower bound, then it will be thresholded. This provides more flexibility and simplicity in color tracking.

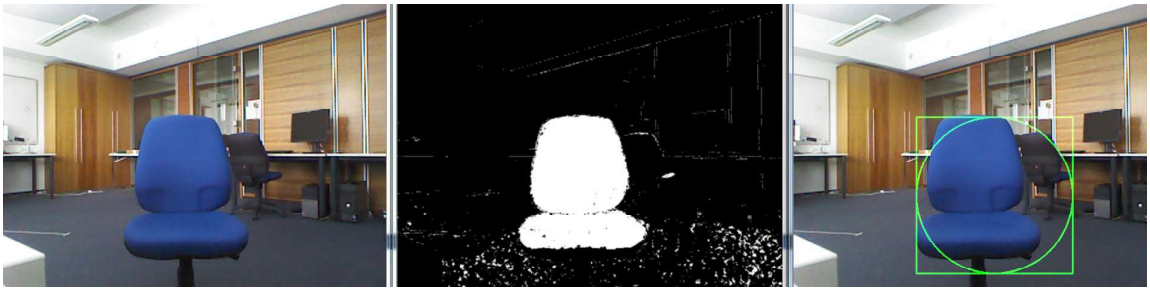

Figure 5.2: Tracking blue color

5.2 Object Detection Using Haar-like Features

OpenCV Haar features detector uses a method that Paul Viola and Michael Jones published in 2001. The features they used are based on Haar-like features that encode the existence of oriented contrasts between regions in the image [22].

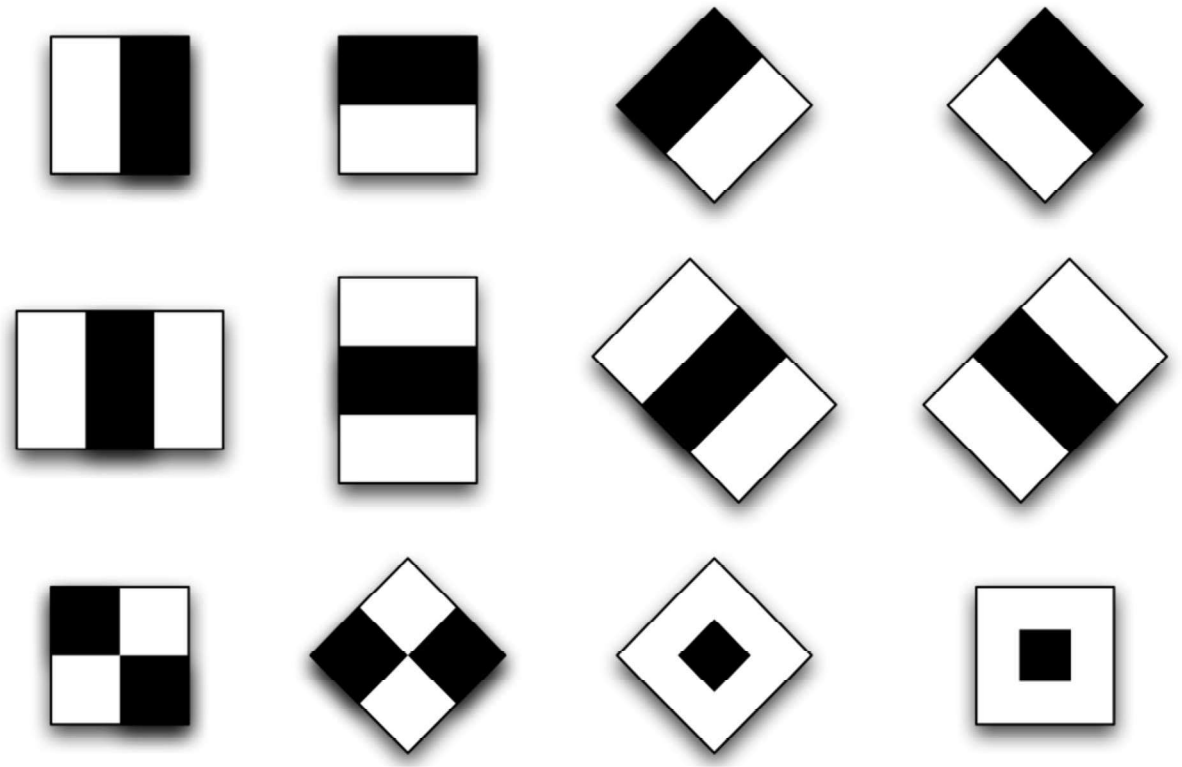

Figure 5.3: Haar-Like Features

To detect an object, first, a classifier working with haar-like features is trained with a few hundreds of sample views of the object (i.e., a face or a car). After a classifier is trained, it can then be applied to a region of interest. To search for the object in the whole image, the classifier is designed so that it can be easily resized, in order to be able to find the objects of interest at different sizes. Hence, or finding an object of an unknown size in the image the scan procedure should be done several times at different scales.

The resultant classifier consists of several simpler classifiers (stages) that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed [22].

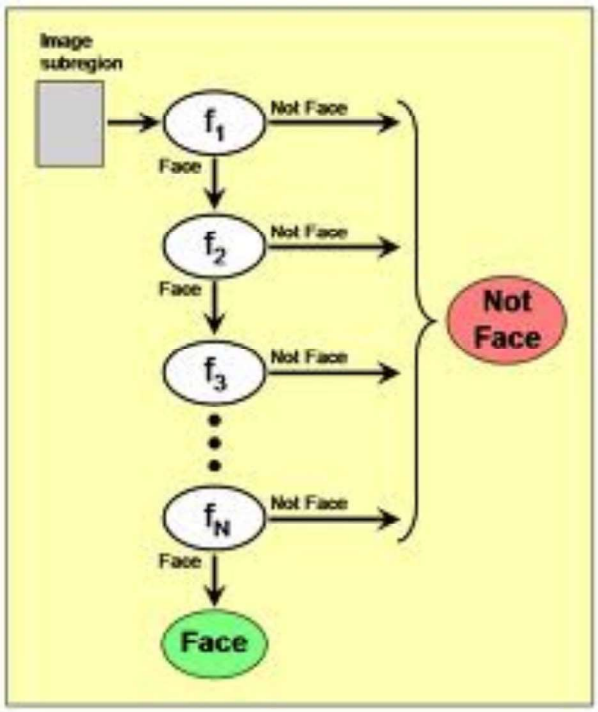

Figure 5.4: Face dedection Flowchart

There are several Haar cascade detectors in OpenCV. The best choice will depend on one's application.

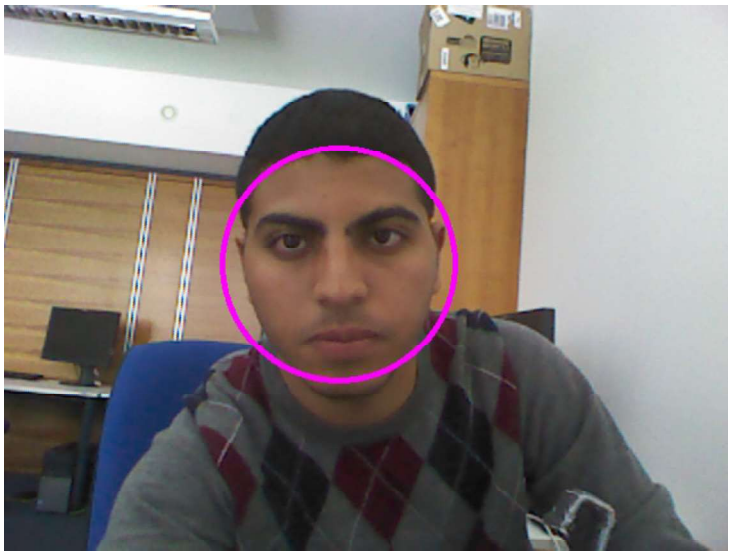

Figure 5.5: Face Detection

5.3 Edge Detection

The edge detection technique implemented by openCV is called canny edge detection, after the name of John F. Canny who developed the the algorithm in 1986. Canny algorithm aims to satisfy three main criteria [19]:

- **Low error rate**: which means the detection of only the existent edges.
- **Good localization**: The distance between real and detected edge pixels is minimal.
- **Minimal response**: Only one detector response per edge.

The algorithm is divided into several steps that should be implemented in sequence, in order for the detector to work efficiently.

Step one: Converting the input RGB image into a gray scale one. This step is only to simplify the algorithm, as the image will be reduced to only one channel on which we will operate.

Step two: Filtering the noise using a Gaussian filter. Where the image is convoluted with the following 5x5 Gaussian kernel matrix:

$$\frac{1}{159}\begin{pmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{pmatrix} \tag{5.1}$$

The convolution is simply sliding the kernel matrix over the image. Generally, starting at the top left corner, so as to move the kernel through all the positions where the it fits entirely within the boundaries of the image. The value of each output pixel is calculated by multiplying together the kernel value and the underlying image pixel value for each of the cells in the kernel, and then adding all these numbers together.

I_{11}	I_{12}	I_{13}	I_{14}	I_{15}	I_{16}	I_{17}	I_{18}	I_{19}
I_{21}	I_{22}	I_{23}	I_{24}	I_{25}	I_{26}	I_{27}	I_{28}	I_{29}
I_{31}	I_{32}	I_{33}	I_{34}	I_{35}	I_{36}	I_{37}	I_{38}	I_{39}
I_{41}	I_{42}	I_{43}	I_{44}	I_{45}	I_{46}	I_{47}	I_{48}	I_{49}
I_{51}	I_{52}	I_{53}	I_{54}	I_{55}	I_{56}	I_{57}	I_{58}	I_{59}
I_{61}	I_{62}	I_{63}	I_{64}	I_{65}	I_{66}	I_{67}	I_{68}	I_{69}

K_{11}	K_{12}	K_{13}
K_{21}	K_{22}	K_{23}

Figure 5.6: Convelution by a Kernel

In the previous figure, the value of the bottom right pixel in the output image will be given by:

$$O_{57} = I_{57}K_{11} + I_{58}K_{12} + I_{59}K_{13} + I_{67}K_{21} + I_{68}K_{22} + I_{69}K_{23} \tag{5.2}$$

If the image has M rows and N columns, and the kernel has m rows and n columns, then the size of the output image will have M - m + 1 rows, and N - n + 1 columns.

Mathematically we can write the convolution as:

$$O(i,j) = \sum_{k=1}^{m}\sum_{l=1}^{n} I(i+k-1, j+l-1)K(k,l) \tag{5.3}$$

where i runs from 1 to M - m + 1 and j runs from 1 to N - n + 1.

Step three: Finding the gradient strength of the image. This is done firstly, by finding the gradient of the image in the X direction Gx, and the gradient of the image in the Y direction Gy. Then calculating the resultant of both.

Gx and Gy are calculated by applying the following convolution masks.

$$G_x = \begin{pmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{pmatrix} \tag{5.4}$$

$$G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{pmatrix} \tag{5.5}$$

The resultant gradient G representing the edges is given by:

$$G = \sqrt{{G_x}^2 + {G_y}^2} \tag{5.6}$$

Step four: Thresholding, which is suppressing all pixel values that have a lower gradient value than the fixed threshold value. This means that edge candidates will only appear depending on the selection of the proper threshold value.

The figures below demonstrate the output edges on the RGB and Depth strams, for different threshold values.

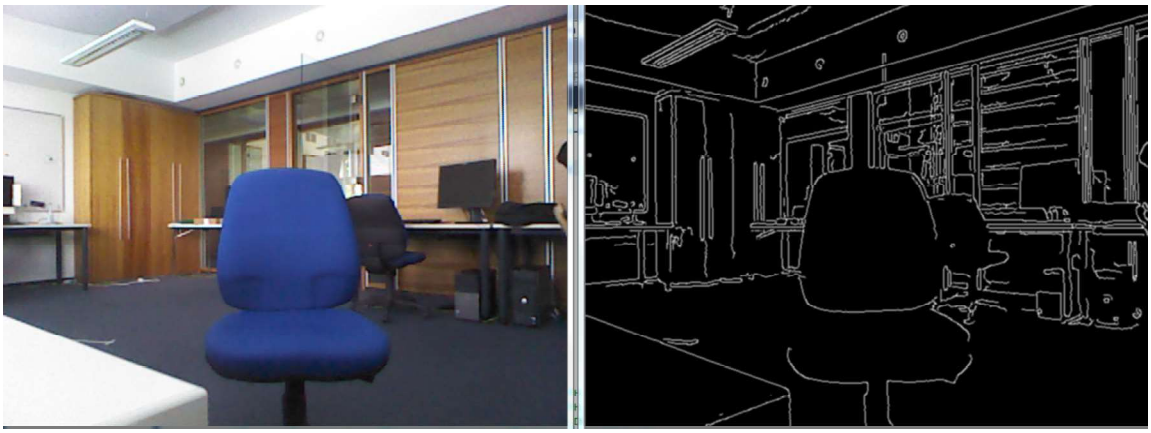

Figure 5.7: Edge detection with threshold 50 on RGB image

Figure 5.8: Edge detection with threshold 50 on Depth image

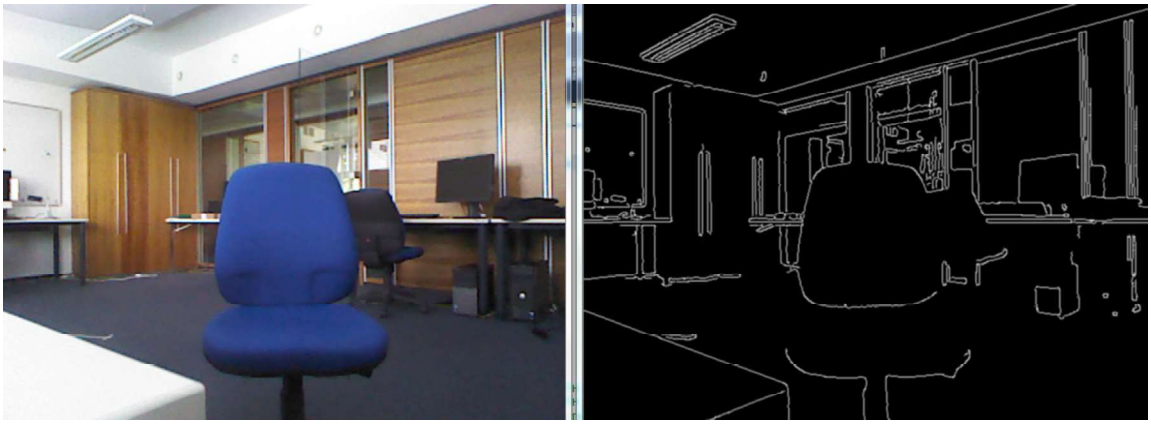

Figure 5.9: Edge detection with threshold 100 on RGB image

Figure 5.10: Edge detection with threshold 100 on Depth image

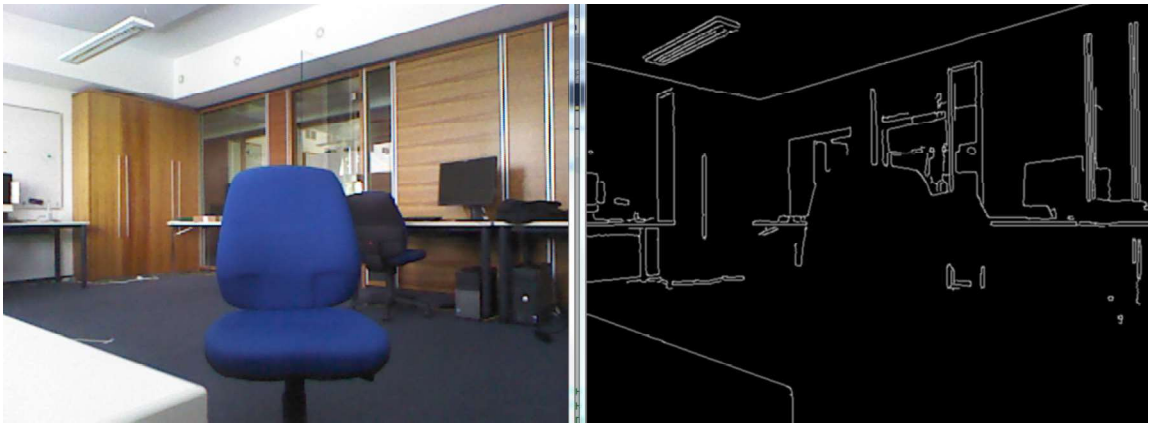

Figure 5.11: Edge detection with threshold 200 on RGB image

Figure 5.12: Edge detection with threshold 200 on Depth image

After testing the algorithm, we selected the threshold value for our application to be: **100**.

5.4 Detection by Active Contours

Object detection through active contours is a simple object detection technique that depends totally on Edge detection. The Idea behind the algorithm is to convert the edges of the objects into a different representation called contours. OpenCv has the ability to detect the location of each continuous contour, and has the ability to draw a bounding rectangle around each one, making them more visible for the user.

The Algorithm was tested on the active contours of both the RGB and depth image streams, with the selected threshold of 100, and results are shown in the figure below:

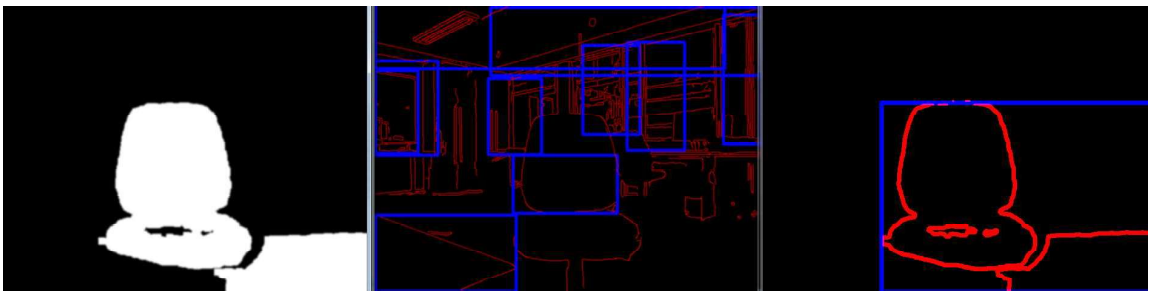

Figure 5.13: Contours detection on RGB and Depth image

As you can observe, the contours on the depth stream provide a better detection for objects of interest, because of the fewer number of detected edges. However in both cases the algorithm failed to provide the required results. The reason is that, the edge detection in both cases did not provide complete continuous edges around each object. Therefore, openCV inferred a series of discontinuous contours for every object of interest, resulting in an inaccurate multiple representations of all objects in the scene.

5.5 Blob Detection

Blob detection is the art of detecting regions of the image that share common features different from that of the surrounding background [11]. In other words, it is an algorithm used to determine various groups of connecting pixels that are related to each other. This is useful for identifying separate objects in a scene, or counting the number of objects, etc..

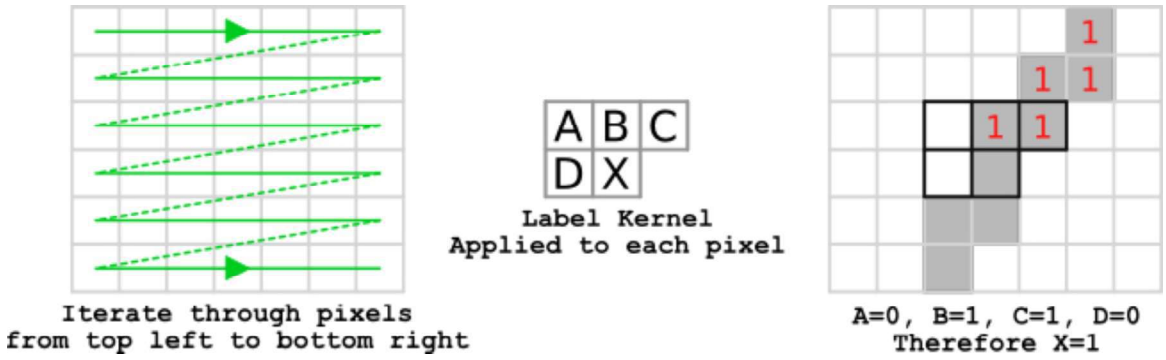

Figure 5.14: Blob detection Technique

As shown in the algorithm below, the input image is first thresholded, therefore separating it into foreground containing objects of interest, and background. Then, the algorithm goes through each pixel in the image from top to bottom, applying a label kernel similar to the one in figure 5.14.

Blob Detection Algorithm
l = 1
for each pixel
if pixel X is foreground
if neighbours A,B,C & D are unlabelled (equal to zero)
label pixel X with
increment l
else
num = neighbour label A,B,C & D with least value, not including 0
label pixel X and pixels A, B, C & D if foreground with num
end if
end if
done

The algorithm runs until all pixels are labeled. Finally, blobs are identified as groups of pixels having the same label.

The Blob detection library we used in our research is 'cvBlobsLib'. The Library was registered on SourceForge.net on Oct 8, 2009, and is described by the project leader as an openCV extension to find and manage connected components in binary images (http://sourceforge.net/projects/cvblobslib/).

Some methods from cvBlobsLib were used to detected the blobs in the depth binary image of the kinect. The output blobs were marked by a bounding rectangle around them, and the results were as follows:

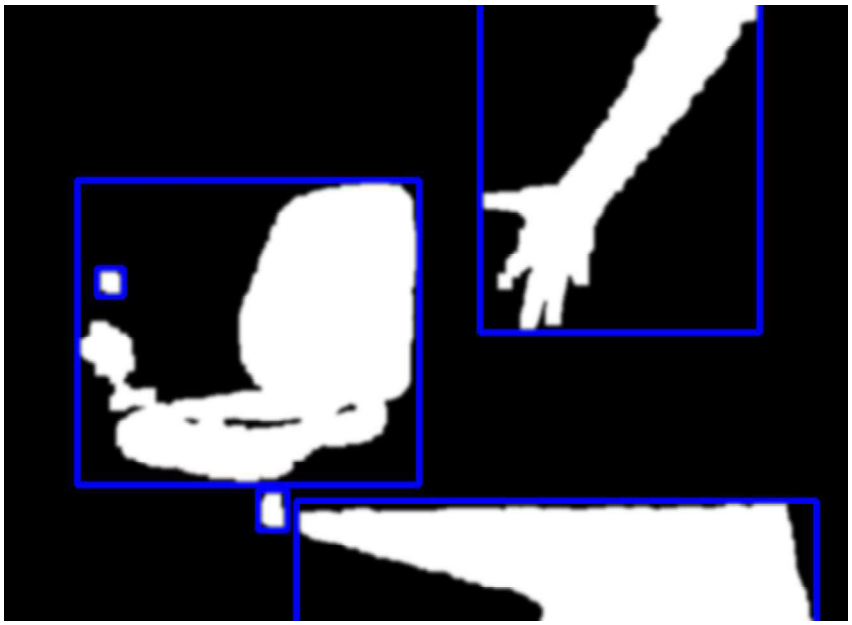

Figure 5.15: Blob detection

5.6 Summary and Conclusion

Throughout this section we reviewed the results of different object detection techniques that we proposed. Starting with color detection that proved its effectiveness in segmentation. However it is constrained to the detection of one color at a time, which is not acceptable for our application. Then, we moved to the technique of Haar-like features detection, and we understood how it works. But again, it constrained us to the openCV cascade detectors that only focus on human features, such as eyes, mouth, face and body. After that, we tested the idea of edge and contours detection, which unfortunately, lead to unacceptable results. Finally, was the proposal of Blob detection, which yielded out the best results among all the previous techniques, and was acceptable up to a very great extent with respect to our application.

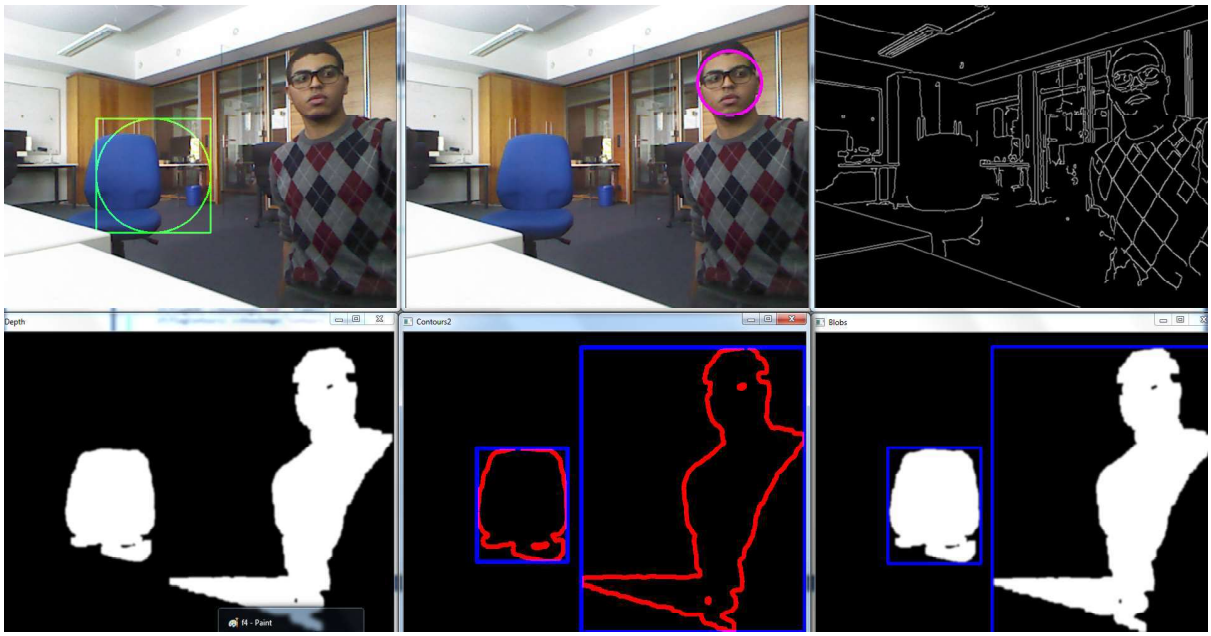

Figure 5.16: Object detection review

To conclude, for applications of generic obstacle avoidance or tracking, that do not focus of avoiding or tracking obstacles of certain specific shape or color, Blob detection is very recommended.

Chapter 6

Path Planning

The general frame work of path planning could be summarized as follows: given a description of the environment, generate a feasible, geometric path from the actual location to goal. This process of directing the robot includes intermediate tasks and assignments that must be completed for the robot to find a collision free path leading to destination.

FALTER uses the technique of situational path planning to explore the environment ahead. This chapter explains the algorithm behind the short term planner, and how it was structured to deal with the input from the depth sensor of the kinect.

As previously mentioned, the visibility range of the kinect has been reconfigured to be within $0.9 - 3.0m$. Subsequently, the situational path planner will plan for a path that starts $0.9m$ in front of the camera and ends 3.0 m away from it. The algorithm recomputes the path with every depth stream from the kinect, thus the path is re-planned 15 times every second.

6.1 Find Safe Area

For the quadrocopter to configure a path, it needs information regarding its starting position, its target position and obstacles on the way. In our case, since the kinect is installed on the copter, the starting position is $0.9m$ in-front of the kinect camera and in the middle of the streamed image frames.

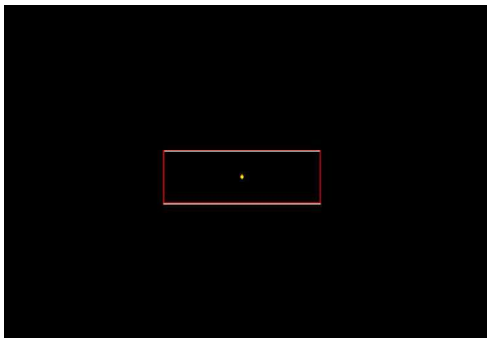

Figure 6.1: Start Position

Our algorithm locates the target position by scanning the scene for an empty spot three meters away from the current copter position. To find such a potential spot, first we need to know the copter dimensions. In reality, the copter dimensions and the obstacle dimensions are fixed. From the point of view of the camera however, the further the objects are from the camera, the smaller they seem. To simplify this problem, we assumed that the copter dimensions relative to the depth are constant. Knowing that this assumption will not affect or invalidate the algorithm, because the planner algorithm recomputes, and thus re-corrects, itself 15 times per second. The chosen copter dimensional configuration is a rectangle with 150 pixels of width, and 50 pixels of height.

To minimize computational costs, the algorithm scans the frame in the way shown in figure 6.2.

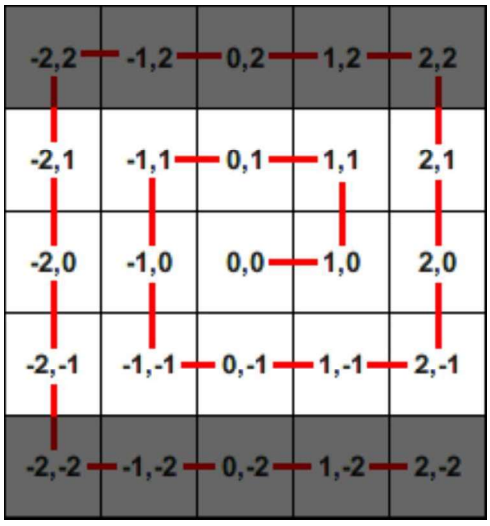

Figure 6.2: Spiral Loop

Starting from the middle of the frame, and with a rectangle with the size and di-

mensions of the copter, the algorithm performs a spiral scanning operation. Each step taken by the spiral loop represents a new possible target location for the copter. With each step in the spiral loop, the algorithm checks the validity of the new position by checking all the $(150x50)$ pixels of the rectangle representing it. Since the input image is binary, this implies that if one or more pixels of the new location are non-zero, then this location contains all or part of an obstacle, and is not valid. The algorithm keeps on running until it finds the first $(150x50)$ rectangular area, where all the pixels are zeros. This location then becomes the target position that is ought to be reached by the path planning algorithm.

The figure below shows how the algorithm reacts and finds the target locations for different binary scenes:

Figure 6.3: Target position allocation for different scenes

Since this algorithm is constructed for flying objects, it is very dangerous to allow the target location to change suddenly between frames. That is why two safety measures have been taken into consideration while structuring the algorithm. First, some adjustments have been made so that the spiral loop of the new frame starts running, not from the middle of the screen like the very first loop, but from the end point of the loop in the frame before it. This way assures very small variations in target allocation from one frame to another, as it forces the algorithm to find the closest empty area to the one it has already located in the frame before.

The second safety measure taken to smoothen the transition between frames, is through applying a simplified version of the kalman filter. The basic concepts and detailed implementation of the kalman filter is presented in [10]. Kalman filter is an optimal estimator that predicts the next state based on information about the previous state and the current control signal. From an abstract point of view, the kalman filter could be considered as a black box with two control inputs: sensor variance (Q) and control variance (R). Q and R could be descried as the uncertainties in sensor and control input readings, or in other words, as to what extent are the control input and sensor readings trustworthy.

Since the kinect does not take a control input, the control variance R could be ignored. The sensor variance Q however, could be tuned to achieve desirable results. If Q is zero,

the algorithm will behave as if there is no filter. As we increase Q, indicating less trust in the algorithm, system convergence to the target position will be slower, and thus the transition between frames becomes much smoother.

Figure 6.4, shows how the algorithm behaves after applying the two safety measures:

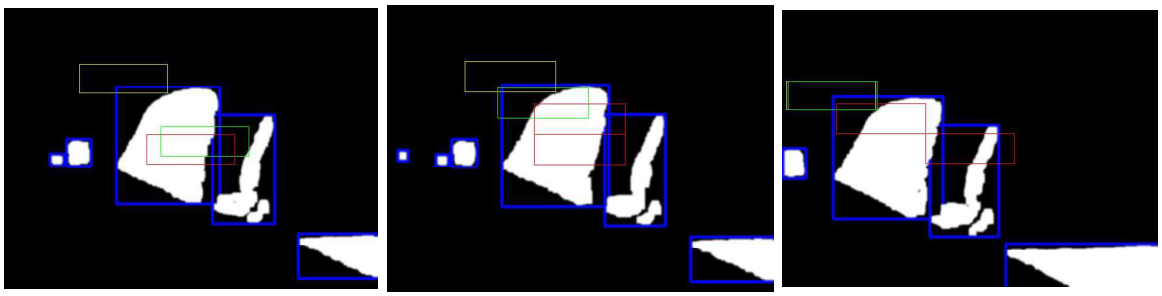

Figure 6.4: Target position allocation for different scenes

Yellow rectangle represents system behaviour before filter, Green rectangle represents system behaviour after filtering and Red rectangles represent how the target is being reached.

6.2 Path Planner

The path planner of FALTER relies on the basic concepts of the A* algorithm. This is a situational path planner that constructs a three dimensional path in space. As mentioned above, the path planner is a short term planner, and does not plan any further beyond the visual capabilities of the kinect. The path is computed once every depth frame, and keeps modifying as the kinect changes position or orientation.

The A* algorithm is a path finding algorithm that finds the least cost path while keeping a sorted priority queue of path segments along the way. Typical costs could be time, distance, velocity or acceleration. In our case we consider the path with the least distance traveled to be the optimal path. Our cost function (F) is the sum of two functions:

- The G **cost** function, which is the resultant distance from the starting position to the current node, along the taken path.
- The H **cost** function, which is the heuristic estimate of the resultant distance from the current position to destination.

Equations 6.1, 6.2 & 6.3 indicate how the G, H and Fcosts are calculated. X_{dest}, Y_{dest}, Z_{dest} represent the 3D coordinates of the target postion, and X_{curr}, Y_{curr}, Z_{curr} represent

the 3D coordinates of the current postion, and $G_{cost}Curr$ is the G cost of moving from the start node to the current one along the taken path.

$$G_{cost}New = G_{cost}Curr + \sqrt{(X_{dest} - X_{curr})^2 + (Y_{dest} - Y_{curr})^2 + (Z_{dest} - Z_{curr})^2} \quad (6.1)$$

$$H_{cost} = \sqrt{(X_{dest} - X_{curr})^2 + (Y_{dest} - Y_{curr})^2 + (Z_{dest} - Z_{curr})^2} \quad (6.2)$$

$$F_{cost} = G_{cost} + H_{cost} \quad (6.3)$$

To compute the optimal path, the algorithm divides the scene into a grid of nodes. Each node corresponds to a potential copter position, thus occupying an area equal to the copter area ($150x50$ pixels). Nodes are typically represented by the following set of data defining them:

- The position of the center point (x,y) of the node with respect to the pixel frame coordinate system, shown in figure 6.5.
- The 3D coordinates of the center point of the node in meters, with respect to the kinect frame of reference, shown in figure 6.6.
- Node index
- Node's parent index
- G_{cost}
- H_{cost}
- F_{Cost}

The algorithm sequence starts from the current position of the copter, presented by the node in the middle of the frame, having center (x,y) coordinates of (0,0). This node is then added to a priority queue called closed list, and is set to be the parent of its eight adjacent nodes.

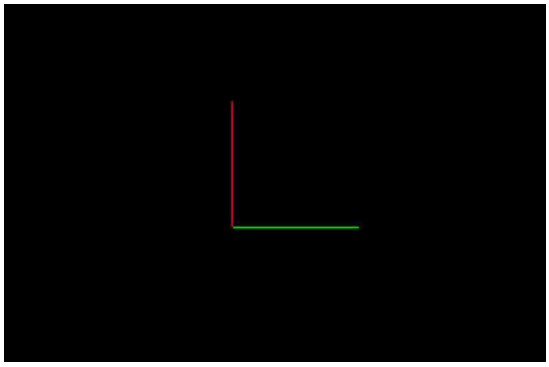

Figure 6.5: 2D coordinate system

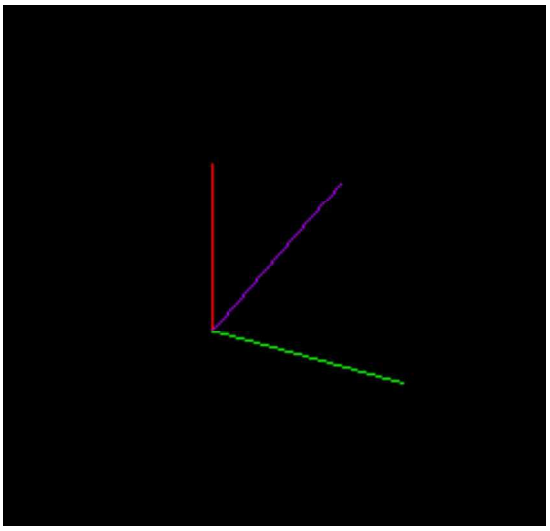

Figure 6.6: 3D coordinate system

For each of the eight adjacent nodes, figure 6.6, the G, H and F costs are calculated, then they are added to a different priority queue called open list. The node with the least F_{cost} is then removed from the open list, added to the closed list, and is set to be parent to its eight adjacent nodes.

The process keeps on repeating itself until the target position is reached, and gets added to the closed list. As shown in figure 6.7, The path is then inferred by tracing backwards through the closed list starting by the last node, and following its parent and parent of parent, and so on until the start node is reached.

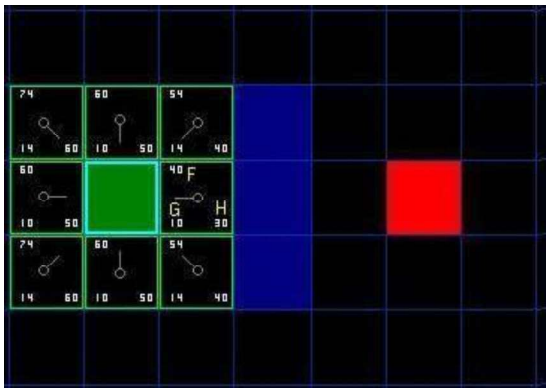

Figure 6.7: Checking all adjacent Nodes

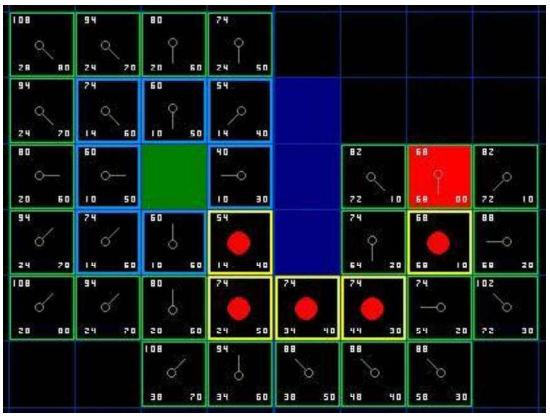

Figure 6.8: A* path calculation

The path is a set of nodes from the current position to destination, and each node contains information regarding its 3D coordinates in the real world. Thus, the path is considered to be a set of 3D coordinates allowing navigation in 3D space. A typical

resulting path is shown in the figure below.

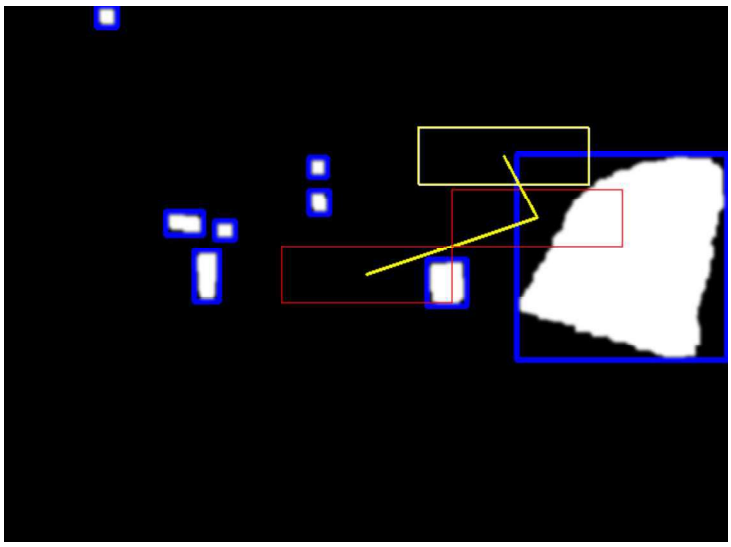

Figure 6.9: Path planner Result

To smoothen up the path and to provide more continuity to it, a 3D cubic interpolation was performed over the resulting set of path points.

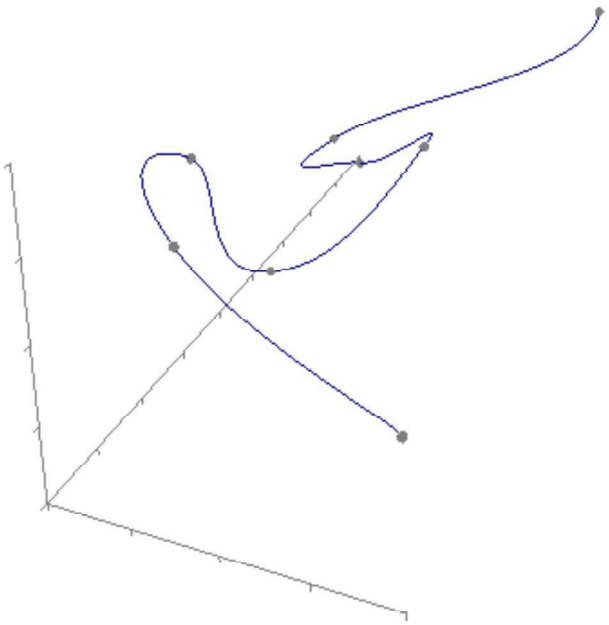

Figure 6.10: 3D cubic Interpolation

The cubic interpolation is done between two points of interest, and two points on either side of them. So the function needs four 3D points labeled p_0, p_1, p_2 and p_3. Our interpolation algorithm creates new points between our target points, by executing the following equation 10 times.

$$a_0 {m_u}^3 + a_1 {m_u}^2 + a_2 m_u + a_3 \tag{6.4}$$

Where m_u is within the range 0-1 and is incremented by 0.1 after each execution. The coefficients a_0, a_1, a_2, a_3 are as follows:

$a_0 = p_3 - p_2 - p_0 + p_1$
$a_1 = p_0 - p_1 - a_0$
$a_2 = p_2 - p_0$
$a_3 = p_1$

After performing the algorithm the resultant path (red curve) seemed more smooth and reliable, as seen in the figure below:

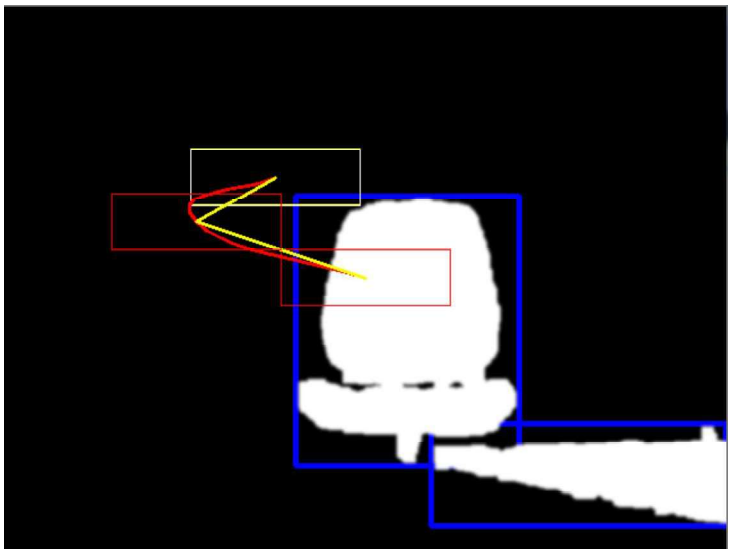

Figure 6.11: Resulting Path after Interpolation

6.3 Algorithm Testing

After being implemented, the algorithm was tested for different scenarios and scenes. The results were satisfactory, and the algorithm reacted perfectly and passed all testing

scenarios. The algorithm reaction to some of those tests is demonstrated in the figures below:

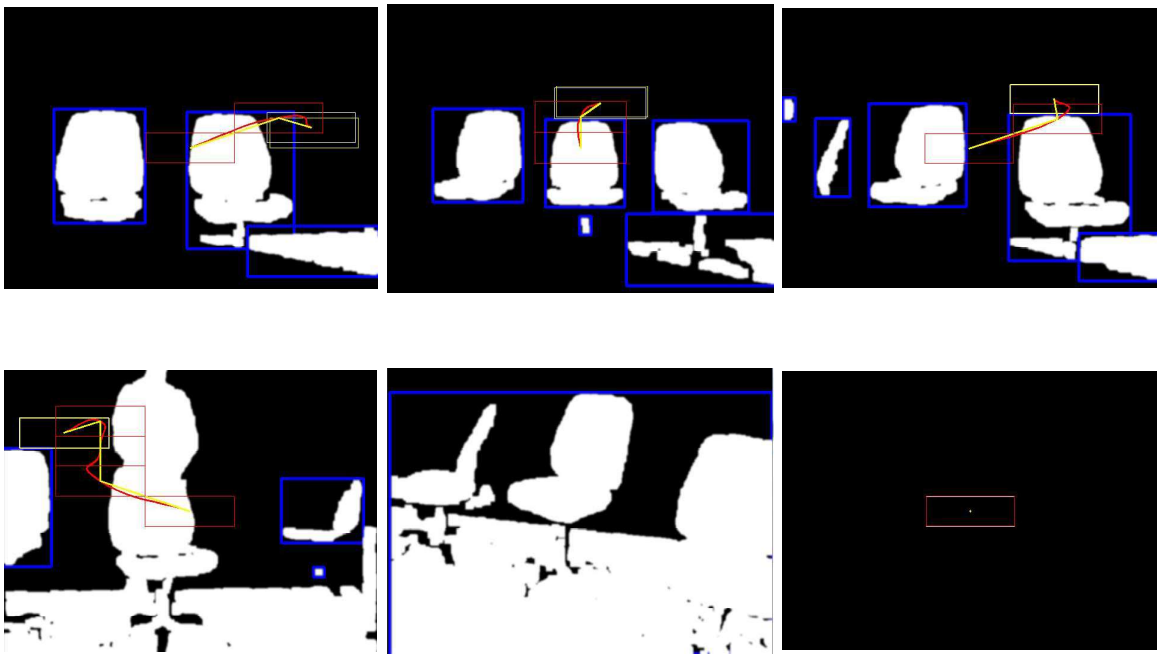

Figure 6.12: Algorithm Reaction to different scenes

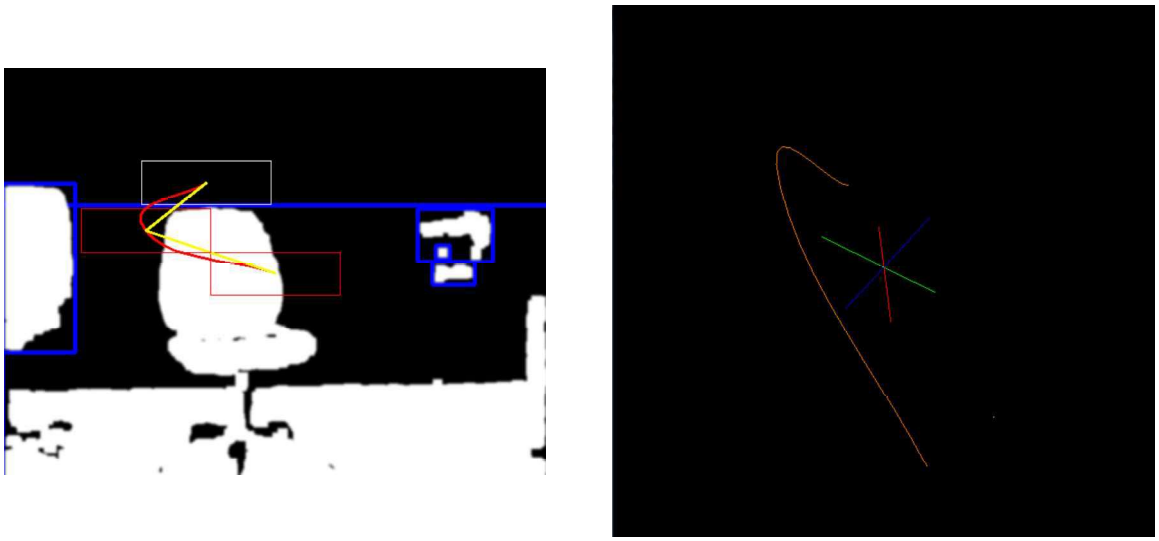

Figure 6.13: Path and Its 3D visiualization

Chapter 7

Virtual Integration

During the past years of research [8] [9], a Matlab simulink simulation was created to imitate the physical behavior of FALTER in a virtual environment. The physical system was modeled with mathematical functions. Every sensor and every actor of the system was modeled in such a detailed way, so that it meets the requirements of its purpose in the simulation model.

For the virtual integration of the FALTER unit, the tool Simulink from The Mathworks, Inc. is used. Simulink is a design and simulation tool that has a graphical user interface for building graphical models of blocks and signals. A block can perform mathematical functions, generate visual output or contain other blocks that perform different actions. The signals represent the data flow between different blocks. Finally, to include the control code that performs the path planning and navigation, Simulink provides S-function blocks, that provide a method to include C, C++ or Fortran code in the simulation.

The developed simulator, shown figure 7.1, can show the quadrocopter from three different viewpoints: falter camera, which traces every moment of FALTER, corridor, which gives a wider glance over the environment, and watch, which shows a top view of the whole environment.

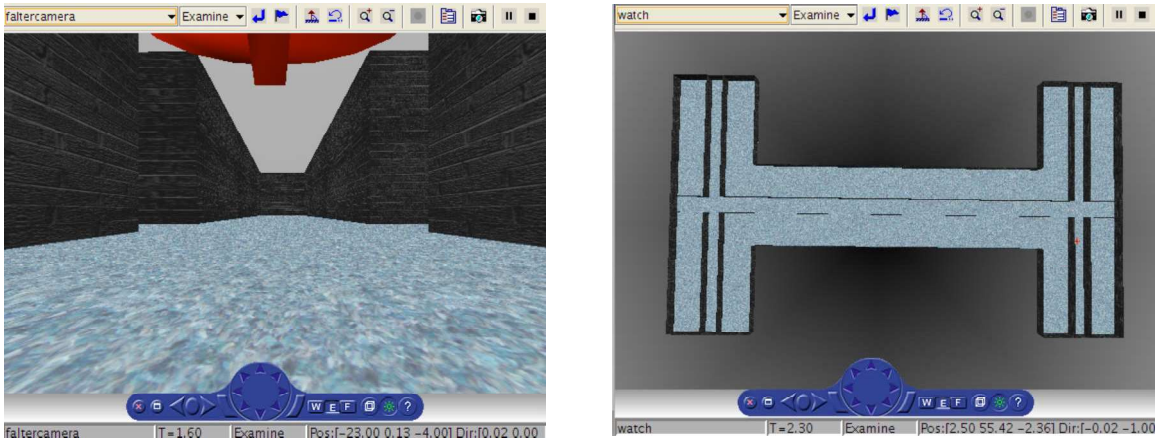

Figure 7.1: Simulation Enviroment

7.1 Replication Of The Depth Sensor

The idea behind the depth sensor model, is to generate a set of beams in the direction where the depth sensor is looking. The depth image could then be formed by determining the distance from the camera to the points of intersection of the beams with obstacles and walls. The number of beams represent the resolution of the depth image. The depth camera model generates a vector of 3072 beams, which is then passed to a Matlab m-function to resize it to our default $640x480$ image. The accurate technique of measuring the distances using beam intersections is presented by Florian Mutter in his thesis "virtual integration" [9].

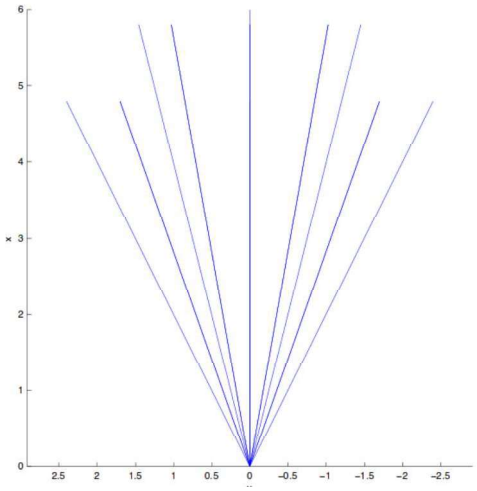

Figure 7.2: Depth sensor beams

Since the depth camera measures the perpendicular distance to the obstacles, the

beams in figure 7.2 are corrected through multiplying them by rotation matrices. Typical rotation matrices (R_x, R_y, R_z) about the X, Y & Z directions are shown below.

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$$

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The depth camera model is designed not to calculate distances of more than $7.0m$. Therefore, as shown in the block digram of figure 7.3, all obstacles positioned further than $7.0m$ are flagged with a value of zero, indicating that they are not seen by the sensor. Then while calculating the distance values, figure 7.4, if the walls indicate zeros, then a distance value of zero is directly returned. This minimizes the wast of computational time and power wasted on calculating distances of undesirable obstacles.

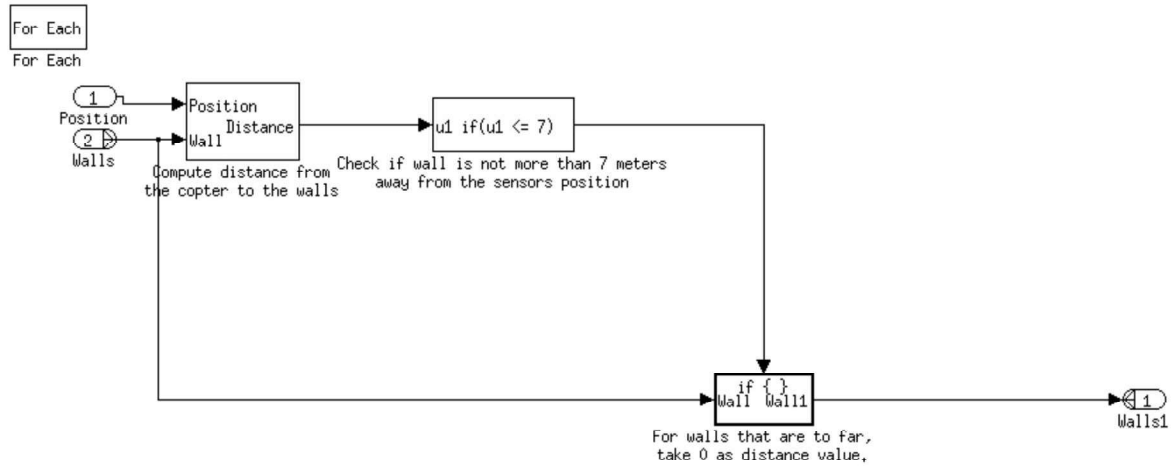

Figure 7.3: Block digram for flaging far walls

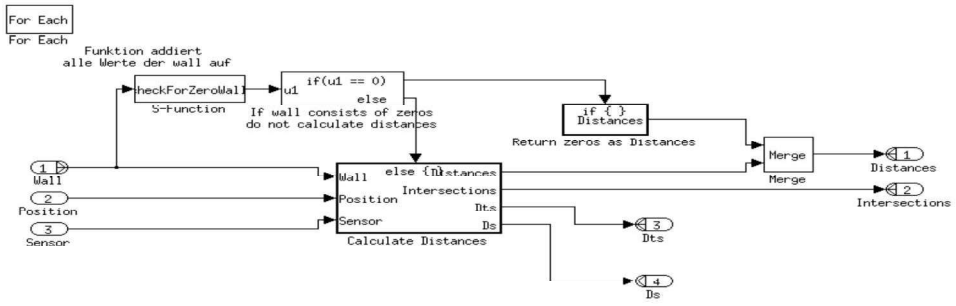

Figure 7.4: Distance calculation block digram logic

The kinect sensor however, has been preset to work within a maximum range of $3.0m$. So to simulate it correctly, the output data from the depth camera model has been post-processed so that any value indicating a distance more than $3.0m$ is overwritten to $3.0m$, and any value indicating a distance less than $0.9m$ is overwritten to $0.9m$. The two figures below show how the output from the depth camera block looks like before and after such processing:

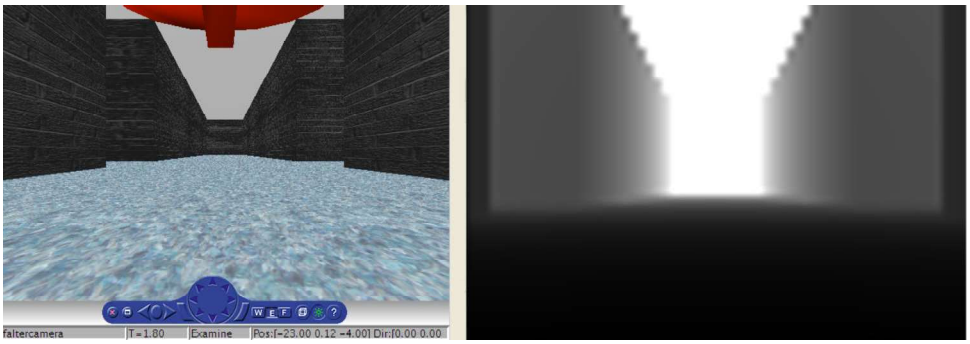

Figure 7.5: Output stream of Depth camera model before processing

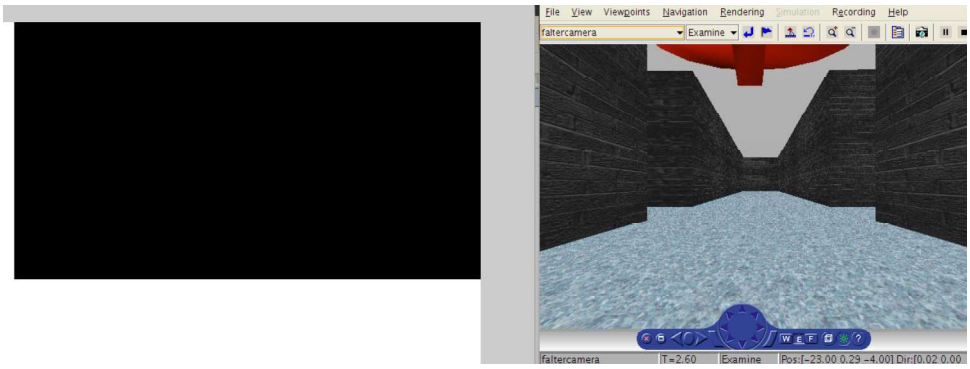

Figure 7.6: Output stream of Depth camera model after processing

7.2 Code Integration

To include the navigation and path planning code developed in C++, matlab S-functions were used. The S-functions are represented in the block model as normal blocks with input and output ports, and are easily connected to other blocks of the simulation.

The only problem in code integration was fitting the set of points generated by the path planning algorithm to match the map representation. The map and the copter have the coordinate system shown in figure 7.7. So, for the copter to fly correctly through the map, the whole set of path points should be transfered from the local copter coordinate system, to the map global coordinate system.

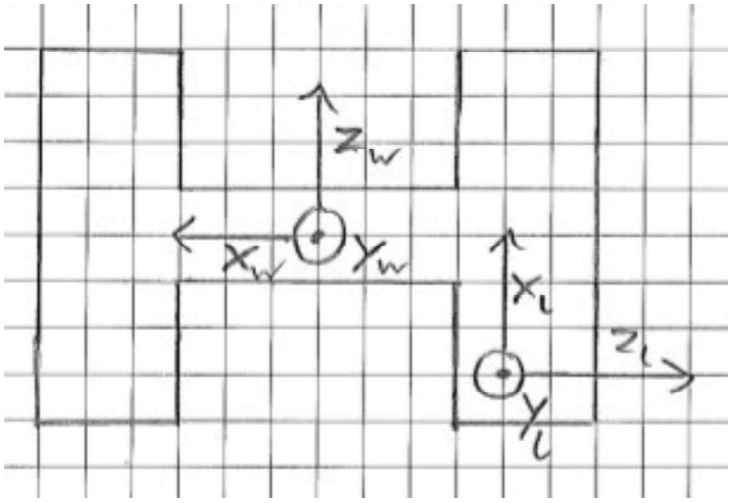

Figure 7.7: Global and Local coordinate systems

The transformation equations are as follows:

$X_{global} = -X_{local}$+copter current X location
$Z_{global} = Y_{local}$+copter current Y location
$Y_{global} = Z_{local}$+copter current Z location

Moreover, The map is divided into a grid, with blocks of $0.2x0.2\ cm^2$ of area. So, each of the X, Y values should be divided by $0.2m$ after being transformed into the new coordinate system.

7.3 Testing and Achievement

The Algorithm was successful integrated with Matlab simulink, and was simulated through the virtual environment. Due to the very high computational costs, the algorithm was

running very slowly, 10 times slower than real time. The copter took off, located the target location, planned the path, converted the path points to the global map frame of reference and followed the path without colliding with any of the walls on the way. The test was considered a success and the algorithm validity was confirmed. Figure 7.8 shows some snapshots from the navigation process.

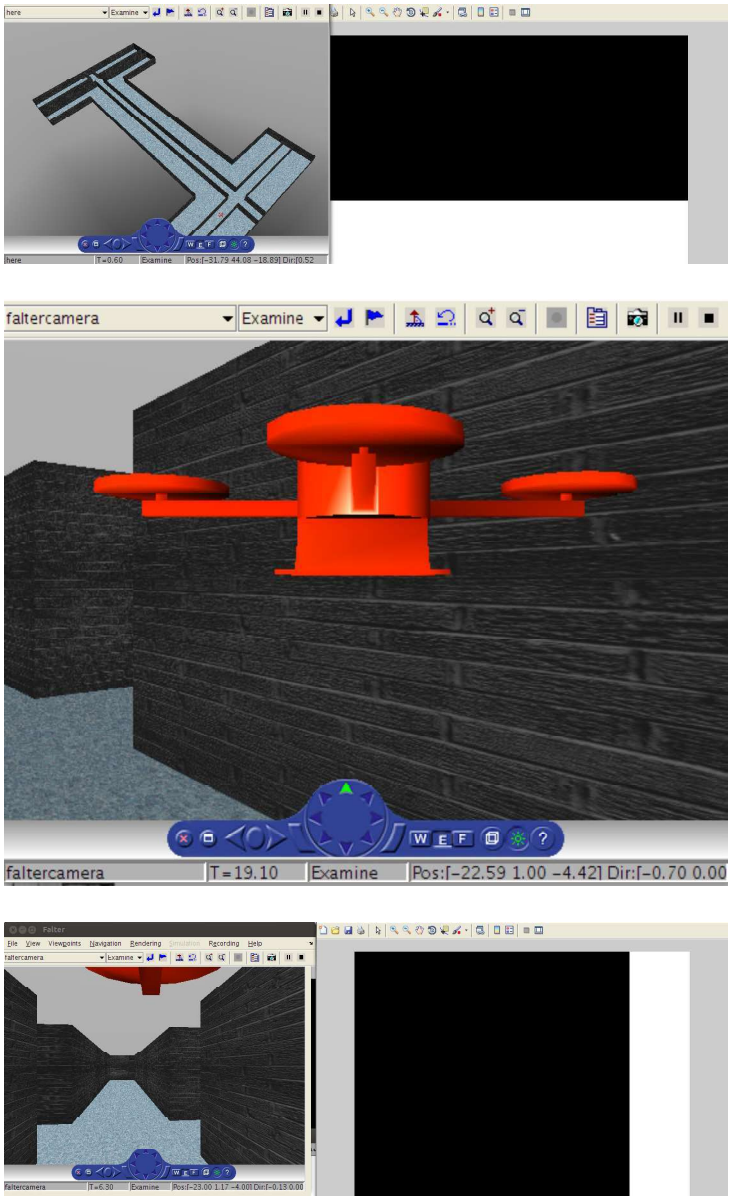

Figure 7.8: Simulation Snapshots

Chapter 8

Further Development

A further step in the direction of development, is to extend the short term path planner descried in chapter 6 to be part of a long term one. The situational path planner is not enough to plan long journeys, because it simply is a goalless algorithm. It just figures out the next best step to take into the environment, without following a plan or without having a target in mind. To create a long term path plan, we need to have a map of the environment we are traversing. In addition, we need to keep track of the current position through continuous localization.

The beauty about using the situational path planner as part of a long term one, is the needlessness of having a detailed map of the environment. While constructing a path, such an abstract map is enough, because the situational path planner will take care of avoiding random obstacles that pop up in the scene. The algorithm sequence of operation goes as follows:

1. Apply the A* algorithm to create an initial path from current position to destination, through the 2D abstract map.
2. Start following that path while keeping a constant track of current position.
3. With situational path planner running in parallel, whenever an obstacle appears in front of the copter, the initial path is corrected so that the obstacle is avoided.
4. Both path planners keep on running until target is reached.

To test the algorithm, the virtual map figure 8.2 is created using openCV graphical capabilities. The map is $640x480$ pixels, and is assumed to be $50mx50m$ in real life. This scaling operation is done by the mapping function :

$$(x - in_m in) * (out_m ax - out_m in)/(in_m ax - in_m in) + out_m in$$

The start location with respect to map ccordinate system, shown in figure 8.1, was assumed to be at pixel location $(320x10)$, and the target location was set to be pixel

location (450x410).

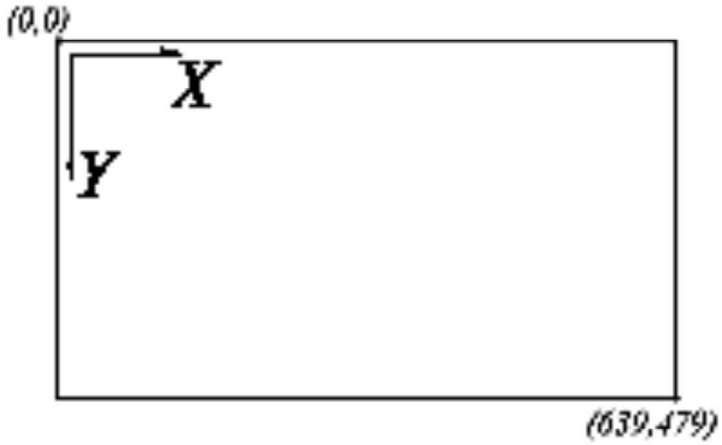

Figure 8.1: Map coordinate system

The A* algorithm was then applied on this map, and the intial path, shown in figure 8.3, was generated.

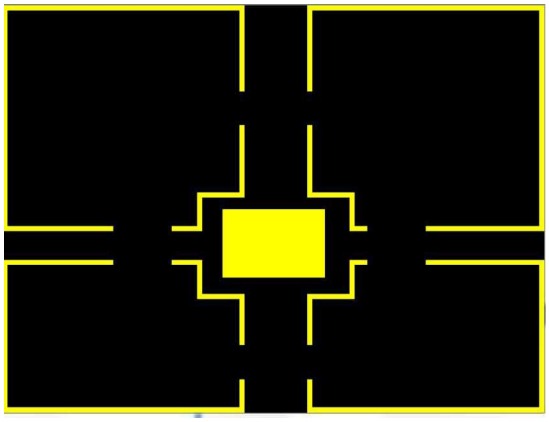

Figure 8.2: Imaginary map

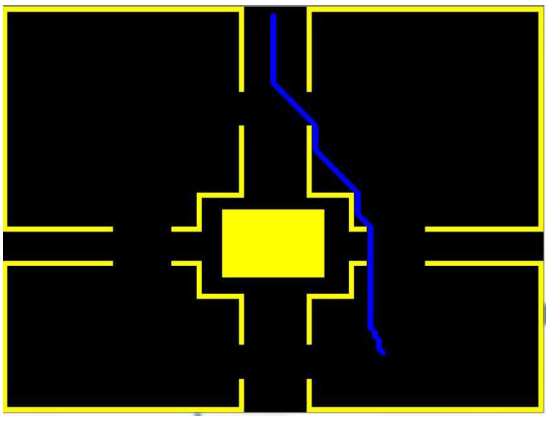

Figure 8.3: Intial path generation

To keep track of the current location, a velocity vector corresponding to the initial generated path is created. As a matter of fact, two velocity vectors are created for both velocity components: V_x in the X direction, and V_y in the Y direction. Both vectors are formed depending on the shape of the intimal path in 2D. The copter is always assumed to be having a constant velocity of $1.5m/s$ in one or both directions. The current location is then inferred through applying the kinematics equation 8.1 bellow, assuming zero acceleration and $0.5s$ sampling time.

$$X_{curr} = V_{intial}t + \frac{1}{2}at^2 + X_{old} \tag{8.1}$$

Assuming that the copter is moving, and no obstacles will appear in the scene, the localization technique was tested. Unfortunately, since the localization technique has the problem of additive error, as shown in figure 8.4, it did not provide accurate localization.

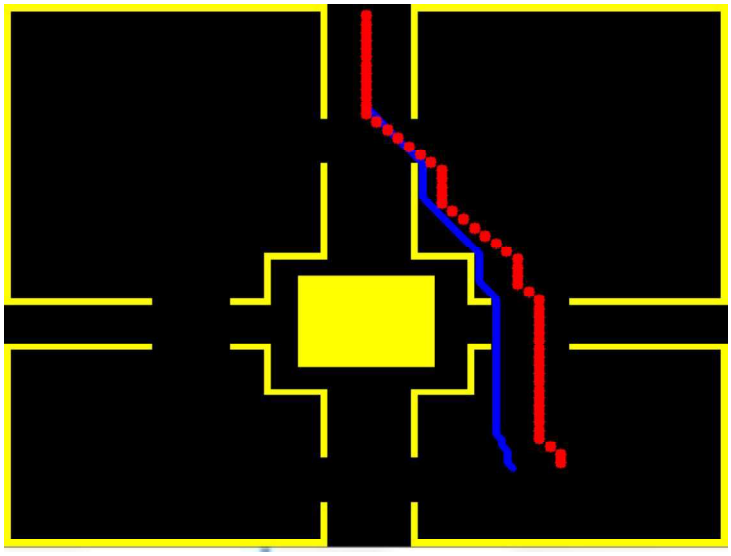

Figure 8.4: Localization

Integrating the long term planning and the situational planning is simply done by adding the current position, inferred from localization, and the readings of the situational path planner. Therefore, correcting the path to avoid obstacles. While developing the algorithm, for simplicity, we did not take into account that the copter can yaw. This constrained the motion of the copter to translation in the forward direction only, as it can not yaw to do sharp or U-turns.

The algorithm was not tested in real time, because of the difficulty to construct an accurate virtual graphical map representing real existing rooms and corridors that will be seen by kinect. Hence, the algorithm validity is still a mater of concern, and needs real time or virtual testing in the future to be validated.

Chapter 9

Conclusion

The aim of this thesis was to investigate new possibilities for navigating a quadrocopter through a complex environment. The hypothesis was that, it is possible to completely depend on the data from a range sensing camera, to plan a safe flight for UAVs through space. To confirm this claim, a kinect depth sensor was used throughout the research for data acquisition and processing.

To plan a safe flight, the quadrocopter must figure out ways to detect objects that block its way. Chapter 5 discussed a variety of different object detection techniques, supported by the object detection library openCV. The approaches were: object detection based on color, object detection using haar-like features, edge detection, contour detection and Blob detection. The results of testing, showed that Blob detection is the technique fitting our application the most.

The second part of the thesis presented the development of a path planning technique, enabling the copter to avoid the detected obstacles. The idea behind the algorithm, was to use the depth map generated by the kinect to locate an empty spot $3m$ away from the copter. Then using the path planning algorithm, a path is generated from current position to the previously located destination. This path avoids all obstacles, while taking into consideration efficiency in terms of the covered distance.

After a series of testing operations for different scenarios, and after examining the algorithm on the FALTER matlab simulation, the results were satisfactory. The copter behavior in the virtual environment indicated that the algorithm reacts in a good way with different levels and complexities in the space it is navigating through. The hypothesis was confirmed, therefore proving the usage of depth sensing cameras to be an effective, efficient and attractive technique in navigating through unknown complex 3D spaces.