NTNU – Trondheim
Norwegian University of
Science and Technology

Visual Sensing in Mobile Robots

Vegard S. Lindrup

Submission date: December 2015
Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Title: Visual Sensing in Mobile Robots

Student: Vegard S. Lindrup

Problem description

Introduction

This project is a continuation of previous projects in developing a concept for robotic maintenance performed by a mobile autonomous robot. Over the course of previous projects, the system has been equipped with a robot manipulator arm, several sensors, a wireless router, on-board power supply based on batteries and a central PC. This equipment is mounted on a steel wagon. The wagon stands on four omni-wheels, each with their own electric motor drivers.

Project Goals

To increase the robot‘s degree of autonomy, it is desired to explore possibilities for reliable and safe movement of the robot without involving a person. The purpose of such a movement may be to reach a docking station or a location where a maintenance or an inspection task will be performed while avoiding obstructions and hazardous situations. As a step towards achieving these goals, the following points shall be carried out:

1. Explore potential methods for autonomous navigation that may fulfill the goals above, where computer vision is the primary navigational aid.
2. Implement one, several or a combination of the methods found in point one. This includes selection and installation of new equipment, e.g. cameras, if necessary.
3. Performance and suitability assessment of the selected implementation with respect to autonomous navigation.
4. Assess how well the system handles errors and potentially hazardous states and situations.
5. Propose changes to the implementation and suggest further work in order to improve the safety and reliability of the system and its ability to navigate autonomously.

Supervisor: Tor Engebret Onshus, ITK

Abstract

Preface

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

List of Figures

List of Tables

Contents

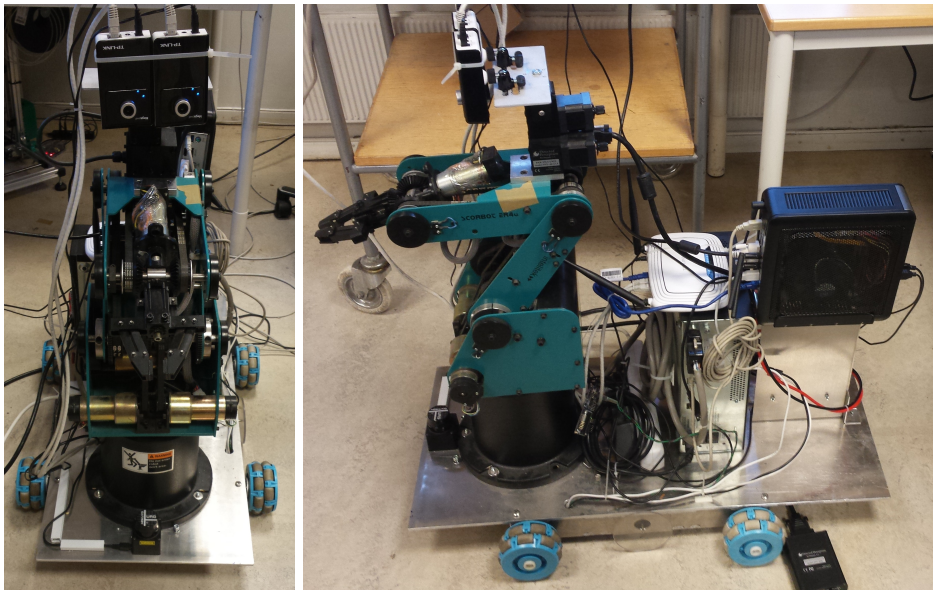
Chapter 1
Introduction

1.1 About this Report

This report presents and documents the work that is done during the ninth semester spesialization project in Engineering Cybernetics, **TTK4550**. The procject is worth 15 credits (studiepoeng), which is approximatly equal to nine full time weeks. The overall learing objectives are that the student shall learn to specialize in a specific field based on scientific methods. Furthermore, the student shall learn to complete a larger project. The final learning goal is to produce a project report. Project work is performed as self-study under the guidence of a supervisor. The project shall be monitored and controlled by a project plan, milestones and progress updates at least once each month.

1.2 Motivation - Mobile Autonomous Robotics and Computer Vision

The field of computer vision has seen an enormous growth over the last few decades - not only in scale, but in accessibility and capability as well. As a consequence of this recent growth, tapping into the field of computer vision is bound to reveal applications that are useful for a mobile autonomous maintenance robot. Recent discoveries within computer vision includes robust feature recognition and object detection, face detection and video processing. The latest great additions to the field are Big Data and Artificial Intelligence, which takes computer vision one step closer to human-like abilities. One of the major benefits of using cameras as sensors are their flexibility. A camera as a sensor can be used for both navigation, mapping and maintenence tasks. Cameras are usefull for remote control as well.



(a) Front view. (b) Side view.

Figure 1.1: The robot used in the project.

1.3 System Overview

The mobile robot being worked on in this project is shown in figure 1.1a. The manipulator arm has been used in previous projects on robotic maintenance, and it was placed on the mobile platform during the master thesis of Petter Aspunvik. This section provides a short description of the systems used in this project and other surrounding equipment. If a more detailed description of the robot and it's equipment is required, consult the thesis of Aspunvik [Asp13].

1.3.1 The Robot

The robot in its current state is the result of several preceeding projects, where the master thesis of Petter Aspunvik [Asp13] and Mikael Berg [Ber13] are the most recent contributions.

Propulsion The steel chassis of the robot stands upon four omni-wheels. The wheel pairs are placed in parallel, making the vehicle uncontrollable along the lateral axis. Each wheel is powered by an electrical motor and motor driver. The motor drivers are controlled with pulse width modulation by an evaluation board from Atmel (Xmega A3BU).

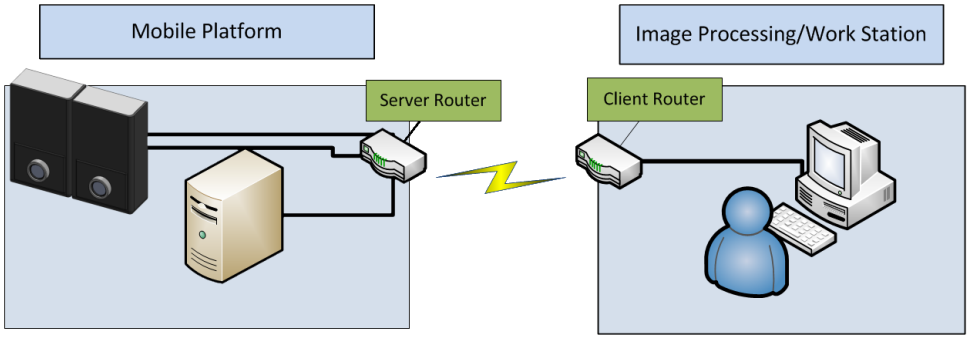


Figure 1.2: Stereo vision set-up.

Sensors The robot has been outfitted with several sensors over the course of previous projects. These are:

- Two odometer wheels with encoders. One on each side.
- Two infrared distance sensors.
- A LIDAR (Light Detection and Ranging).
- Two IP-cameras.

Only the cameras were used in this projects.

Robot Arm A robot arm, SCORBOT-ER 4u from Intelitek, is monted on the wagon. It is intended for educational use in the context of automation and work cells. The robot has five rotation joints plus a servo gripper at the outer joint. Control of the robot in this project is done from the on-board computer on the wagon, which is connected to the robot arm through a USB cable.

1.3.2 Implementation Set-up

Vanishing points and stereo vision are the main topics in this report. The stereo vision portion consist of the two IP cameras, a pair of wireless routers and a remote computer. The remote computer can be any computer which is able to run Open Source Computer Vision Library (OpenCV) and receive video feed from the camera pair. The vanishing point detector is simply a desktop application which accesses a web camera. Both the stereo vision and vanishing point detector applications have been developed with Qt and run on Windows 7.

1.4 Report Structure

How the report is structured, and a very brief description of the contents in each section.

Chapter 2
Background Theory

2.1 Introduction

2.1.1 Chapter Scope

This chapter contains the background theory which is necessary to understand the implementations in chapter 3 and how they are intended to work. Methods for vision based autonomous navigation is also discussed on a preliminary level.

2.1.2 Brief Introduction to Computer Vision

Computer Vision is the field of giving computers the ability to duplicate the way we perceive and understand our surroundings. This is not an easy task for several reasons. Much information is lost in the process of representing a 3d world on a 2d surface. As image processing really is digital signal processing in 2d, it is subject to irreversible loss of information and noise through quantization. In short, the field of computer vision can be summed up to be the science of drawing conclusions about the real world based on pixel values in an image matrix, or a series of such matrices.

2.1.3 Introduction to OpenCV

OpenCV is an open source library with a vast number of advanced computer vision and machine learning algorithms. The library is written in C and C++; it supports Windows, Linux, iOS and Android; and has interfaces to Python, Java and MATLAB. All OpenCV applications in this project uses OpenCV 3.0.0 for Windows. OpenCV for Windows can be downloaded from *sourceforge.net* or "itseez" on GitHub. The download from *sourceforge.net* contains source files, sample programs, sample data and a pre-built library for MSVC 2010 and 2013. The pre-build library can quickly be plugged into an IDE such as Qt Creator or Visual Studio 2013, thus giving the programmer access to all basic OpenCV features. A steb-by-step guide for using both the pre-built and a custom-built library can be found in Appendix A.

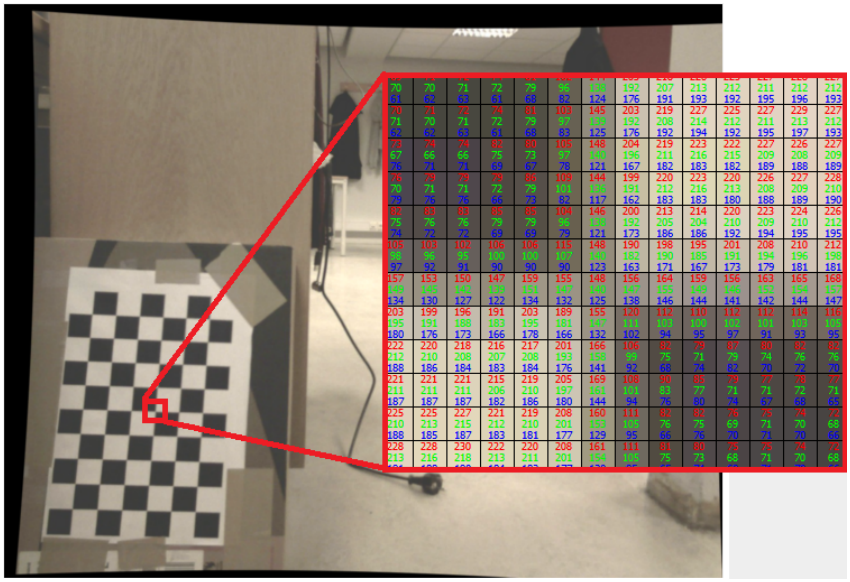


Figure 2.1: Elements in a cv::Mat representing an image. This is a 3-channel image with 8 bit RGB colors.

cv::Mat - The Image Container

2.2 A Brief Introduction to Vision Based Autonomous Navigation

2.2.1 Simultanious Localization and Mapping (SLAM)

Consider a robot that is placed in an unknown environment with no a priori information of the surroundings. If this robot is capable of solving the Simultanious Localization and Mapping (SLAM) problem, it is implied that it can build a map of the surroundings while it determines, simultaniously, where it is located on the same map. This section will, in brief, present some variants of and solutions to the SLAM problem. From [DWB06] a preliminary description of the SLAM problem is based on the following parameters at time k is:

$\boldsymbol{x_k}$**:** "State vection describing position and orientation of the vehicle"[DWB06].

$\boldsymbol{u_k}$**:** Control input at time $k-1$. Used to drive the vehicle to x_k.

$\boldsymbol{m_i}$**:** The position of the *ith* landmark. The true position is assumed to be time invariant.

$\boldsymbol{z_{ik}}$**:** Observed position of the *ith* landmark relative to the vehicle at time k.

$\boldsymbol{X_{0:k} = \{x_0, x_1, ..., x_k\} = \{X_{0:k-1}, x_l\}}$ A set of all previous vehicle locations.

$\boldsymbol{X_{0:k} = \{x_0, x_1, ..., x_k\} = \{X_{0:k-1}, x_l\}}$ All previous control inputs.

$\boldsymbol{X_{0:k} = \{x_0, x_1, ..., x_k\} = \{X_{0:k-1}, x_l\}}$ All previous landmark observations.

Probabilistic SLAM Probabilistic SLAM computes the probability distrobution $P(x_k, m|Z_{0:k}, U_{0_k}, x_0)$ at all time steps k[DWB06], i.e. the joint probability that the robot is in state $\boldsymbol{x_k}$ and that the landmarks can be described by a map,$\boldsymbol{m}$, given historic inputs, landmark observations and vehicle states. The solution algorithm can be implemented in two steps: A time update based on the last updated observation history and current input, and a measurment update where $\boldsymbol{P(x_k, m|Z_{0:k}, U_{0_k}, x_0)}$ is computed based on bayes theorem.

EKF-SLAM The inclusion of Average White Gaussian Noise (AWGN) leads to the application of the Extended Kalman Filter (EKF) to the SLAM problem[DWB06].

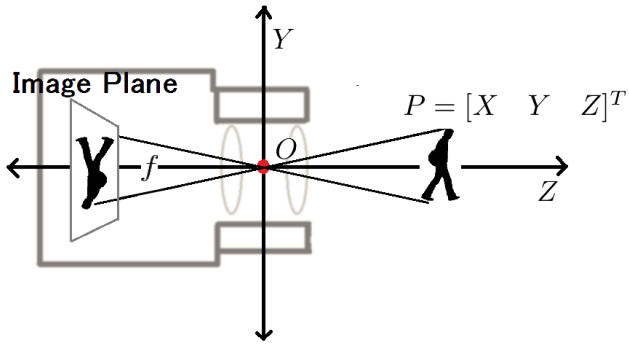
MonoSLAM MonoSLAM by Davison et. al. (2007) is the first successful vision based real-time SLAM algorithm[DRMS07]. The algorithm works by maintaining a probabilistic 3d map of a sparse set of landmarks. The map is continously updated by an EKF. Landmarks are chosen based on som prominent features in the surroundings. In the algorithm, they are represented as flat surfaces with a certain orientation in 3d space. This solution will approximate the fact that the appearence of a feature will change based on the position from where they are observed.

The Kinect Sensor and SLAM The Kinect sensor has been a remarkable contribution to the field of SLAM and computer vision. It is a cheap sensor capable of generating high quality, but noisy, depth maps at $\boldsymbol{30Hz}$. By applying SLAM to the real-time depth data, it is possible to generate a very accurate 3d reconstruction of the surroundings.

2.3 The Pinhole Camera Model

2.3.1 Model Description

Figure 2.2 illustrates the geometry of a pinhole camera. In a pinhole camera, light reflected from some object in a scene will be projected onto an image plane inside the camera house. The



light is projected through the "pinhole", that is the camera projection point $\boldsymbol{O}$, which is located at the origin. Imagine that a single ray enters the camera house through the pinhole, before it hits and exites a sensor element on the image plane. The values retreived from these sensor elements will make up the pixels in an image. The focal length $\boldsymbol{f}$ is the distance from the projection point $\boldsymbol{O}$ to the image plane $\boldsymbol{\pi}$. In a true pinhole camera the image plane will be located behind the lens and the projection point $\boldsymbol{O}$, and the scene projection will be rotated by $\mathbf{180°}$. A virtual image plane placed in front of $\boldsymbol{O}$ is intended to make the figure more straightforward and the mathmatics easier.

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{f_x} & \mathbf{0} & \boldsymbol{c_x} \\ \mathbf{0} & \boldsymbol{f_y} & \boldsymbol{c_y} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix} \tag{2.1}$$

"Learning OpenCV" by Gary Bradski[Bra08] shows how a pinhole camera can be modelled by a 3 by 3 camera matrix $\boldsymbol{M}$ (equation 2.1). This model accounts for two important differences between the ideal and real pinhole camera. First, the imaging chip will often be displaced from the optical center. This displacement is described by the parameters $\boldsymbol{c_x}$ and $\boldsymbol{c_y}$. The second real world problem is that the pixels on the image sensor are shaped as rectangles, not squares. This is accounted for by $\boldsymbol{f_x}$ and $\boldsymbol{f_y}$, the focal lengths in the $\boldsymbol{x}$ and $\boldsymbol{y}$ direction given in pixels. These values are the product of the actual focal length $\boldsymbol{f}$ and size of the individual imager elements $\boldsymbol{s_x}$ and $\boldsymbol{s_y}$. The reason for using these values in the camera model is because $\boldsymbol{s}$ and $\boldsymbol{f}$ cannot be measured without actually dismantling the camera[Bra08].

2.3.2 Camera Distortion

A downside of the otherwise cheap and useful pinhole camera is camera distortion. The distortion is usually severe enough to render the camera useless as a sensor if it is not calibrated. Calibration in OpenCV accounts for radial and tangential distortion by finding five distortion coefficients [3dC]:

$$\boldsymbol{Distortion_{coefficients}} = (\boldsymbol{k_1} \quad \boldsymbol{k_2} \quad \boldsymbol{p_1} \quad \boldsymbol{p_2} \quad \boldsymbol{k_3}) \tag{2.2}$$

where

$$\boldsymbol{p_i} \quad \boldsymbol{with} \quad i \in \{\mathbf{1}, \mathbf{2}\}$$

are the radial distortion constants, and

$$\boldsymbol{k_i} \quad \text{with} \quad \boldsymbol{i \in \{1, 2, 3\}}$$

are the tangential distortion constants.

2.4 Perspectives and Vanishing Points

Vanishing Points A vanishing point is the result of prespective projection. A prespective can be described as the way a 3d world appears when it is projected on a two-dimentional surface. Consider a set of two parallel lines in 3d space that are projected onto an image plane. If these two lines are not parallel to the image plane, their projected representatons will converge to a vanishing point. If the lines are parallel to the image plane, their corresponding vanishing points will be at infinity. The projected lines are formed by the projection lines, that is the direct lines from the parallel lines and the projection center $\boldsymbol{O}$ (figure 2.2 and 2.5). Figure 2.3 illustrates the consept of vanishing points with two parallel horizontal lines.

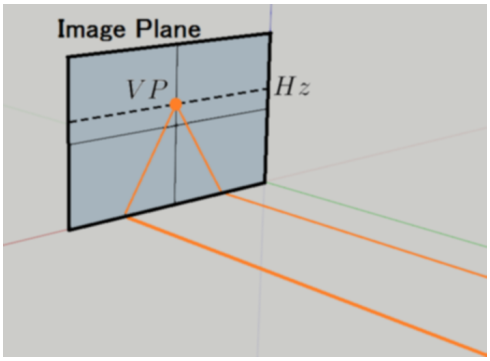


Figure 2.3: Two parallel lines are projected onto an image plane, where they form two lines. These projected lines converge towards a vanishing point on the horizon.

Perspective Transformations When the intrinsic camera parameters (equation (2.1)) and distortion coefficients (equation 2.2) have been calculated, it will be possible to map a point in the outside 3d world to the image plane. This knowlegde allows perspective transformations. A perspective transformation can be used to transform a wall viewed at an angle to an observation point that is perpendicular to the wall. Likewise, the view of a road from a car can be transformed to a birds-eye representation. This birds-eye view can for example be used in combination with a LIght Detecting and Ranging (LIDAR)[Bra08].

2.5 Stereo Vision and Depth Perception

Stereo vision and depth perception is one of the core topics within this report. Here, the theory behind a method using two cameras is presented, while some additional methods are mentioned to provide context.

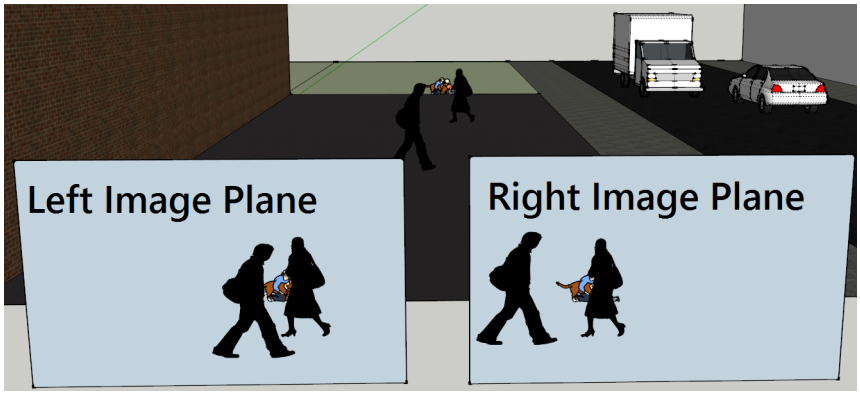


Figure 2.4: Left to right displacement on the image plane based on distance.

2.5.1 Various Methods

Methods for depth perception in computer vision can be separated into two main categories, active and passive[XWS06]. Active sensors will usually project a light pattern onto the scene to be perceived, before sensing how this pattern is displaced by the topology of the scene. The Kinect sensor and 3d-scanners using laser light are typical examples of active sensors. Passive depth perception makes use of many of the same cues we use to perceive depth. The most common passive sensors extract the depth information by observing observing a scene from at least two different positions.

Optical flow is another important method for depth perception. Optical flow may be either active or passive. The passive variant requires only one camera, but depends on motion and a stream of images to extract depth information.[LC12] [Pra80] Observing how much some chosen features in a scene has moved in the image frame at $\boldsymbol{t = 1}$ compared to the frame at $\boldsymbol{t = 0}$ is the basis of depth sensing from optical flow. When the camera moves through a scene where all objects are stationary, objects that are far away will naturally have an optical flow field with a smaller magnitude than objects that are close.

2.5.2 Stereoscopic Vision in General

In this project, passive stereoscopic vision is achieved by using two identical (in theory) cameras placed on the same plane. The gist of passive stereoscopic vision is based on the fact that objects close to the camera pair will have a large displacement from the left to the right camera compared to objects that are further away. This concept is illustrated in figure 2.4.

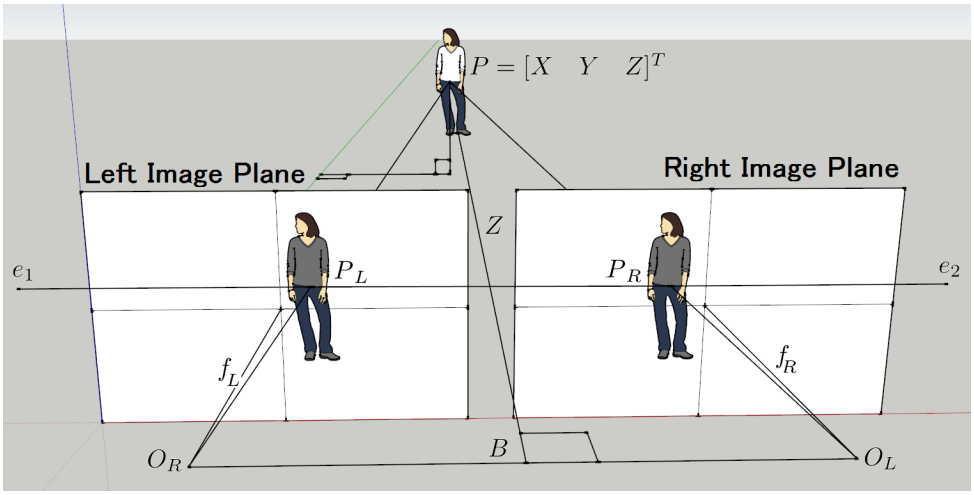


Figure 2.5: Geometry of rectified stereo vision.

Stereo Cameras Figure 2.5 shows an ideal stereo camera model. The model comprise two pinhole camera models where the virtual image planes are located on the same plane. The two image planes are separated by a horizontal translation $\boldsymbol{B}$ which is called the baseline. This implies that the projection point $\boldsymbol{O_L}$ in the left camera, relates to the projection point $\boldsymbol{O_R}$ on the right camera through $\boldsymbol{B}$: $\boldsymbol{O_R = O_L + B}$. Each of the two image planes has a left handed pixel based coordinate system $\boldsymbol{u, v}$, i.e. the origin is in the top left corner and the opposite pixel is in the bottom right corner.

2.5.3 Stereoscopic Vision in OpenCV

The prebuilt version of OpenCV 3.0.0 comes with two stereo matching algorithms: Block Matching (BM) Block Matching (StereoBM) and Semi Global Block Matching (StereoSGBM)[Hir08]. Additional algorithms are available if OpenCV is built with, e.g. CUDA. Both algoritms require a rectified pair of images as input. The basic idea of block matching is illustrated in figure 2.6. The algorithm will search for blocks in the right image that corresponds to similar blocks in the right image. In [Hir08], this process is described as "pixelwise matching of mutual information". This search for mutual information is simplified significantly if the search direction can be limited to

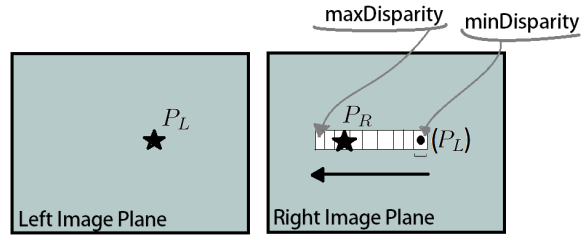


Figure 2.6: Block matching along the epipolar line.

a one dimentional constraint: a horizontal line. This is easy to do, given that the assumption of coplanar image planes holds. The disparity is a measure of how far the matching blocks are displaced from each other.

Chapter 3
Implementation

3.1 Introduction

An obstruction detector and a vanishing point detector are the two attempted implementations presented in this report. The obstruction detector uses stereo vision to perceive depth and distance to possible objects in the path of the robot. The vanishing point detector attempts to find a single vanishing point by detecting lines in the environment before selecting a vanishing point based on line intersections.

3.2 Vanishing Point Detection

3.2.1 Overview

The goal of the vanishing point detector is to provide a setpoint for the robot to steer towards. In other words, steering towards a vanishing point is a good way to reach the end of a hallway or corridor. This was considered to be a good starting point, and possible expantions could be added later. Choosing a method as a basis for a vanishing point detector was not easy. The selected method should be simple, suitable for OpenCV and not go too far beyond the prior knowledge of the author. Another important factor was that spending too much time on this implementation would come at the expense of the obstruction detector. A vanishing point detector method by D. Gerogiannis et. al. [GNL12] showed promise as it was based on line detection, which has good support in OpenCV. The method in [GNL12] appears to be suitable for structured environments with many straight lines, such as hallways, streets and corridors. The steps in the detection procedure are:

1. Detect edges in the image, e.g. by using Canny edge detection.

2. Detect line segments that may be used as vanishing lines based on edges found in step 1. Could be done with the Hough line transform.

3. Filter the detected lines. This is done by modelling new lines by using the major axis of ellipses with very high eccentricity. The ellipses are generated by a split-and-merge algorithm. In short, it will merge similar line segments by assuming that their end points are collinear.
4. Find line intersection points based on the new filtered lines. Each point is stored and assigned a weight.
5. Find the vanishing point among the line intersections based on a voting scheme.

3.2.2 Line Detection

Line detection comprise step 1 and 2 from the list above. OpenCV comes with an implementation of the Canny edge detector ready for use. The detector returns a binary image of the detected edges. Edges are detected by convolving the input image with two kernels $\boldsymbol{G_x}$ and $\boldsymbol{G_y}$. The convolutions will indicate change gradients in the $\boldsymbol{x}$ and $\boldsymbol{y}$ directions which in turn will give the direction of a potential edge. Finally, the detector rejects or accepts potential edges based on two gradient thresholds. Gradients below the lower threshold are rejected, edges above the upper threshold are accepted, and edges between the thresholds are only accepted if their neighbouring gradients are above the upper threshold[can].

At this point, we only have a simple binary image where edges are represented as white pixels on a black background. The next step is to interpret these edges as lines. Line detection is performed by using the probabilistic Hough line transform. This is an already implemented function, which takes in the edge image from the previous step, and return a set of point pairs representing line segments. The benefit of using the probabilistic detector is that it can perceive an edge with a discontinuity as a singe line.

3.2.3 Line Filtering

Line filtering is performed by splitting and merging ellipses until their major axis represents a set of approximations to collinear points.

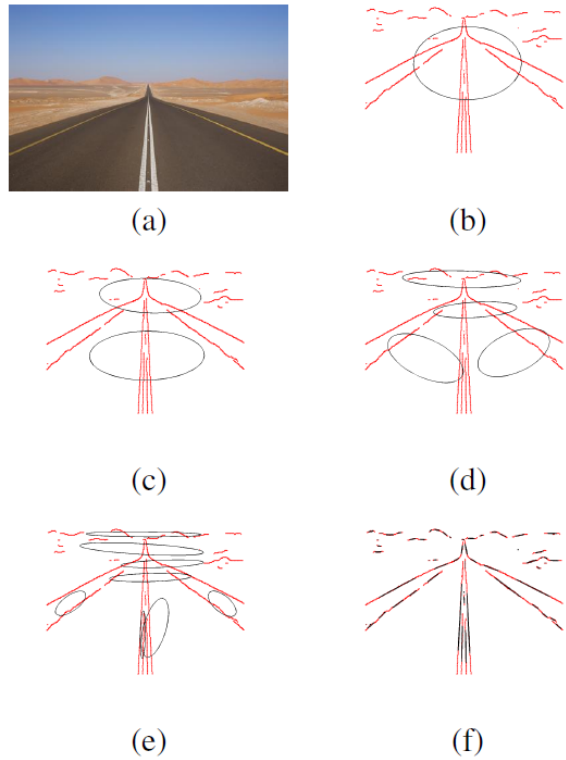


Figure 3.1: Gerogiannis concept of representing lines by eccentric ellipses. This image is taken directly from [GNL12].

The points is the set of points that define the previously detected line segments. This algorithm is called Direct Split and Merge (DSaM), and it is explained in another paper by D. Gerogiannis [GNL11]. Line segments returned from the probabilistic Hough transform will often overlap or be very close to each other. The purpose of this step is to get a cleaner representation of the contours in the environment.

Figure 3.1, taken from [GNL12], illustrates the steps in the DSaM algorithm.

3.2.4 Vanishing Point Detection

When the detected lines have been filtered and stored, they will be passed to the vanishing point detector in the function *getVanishingPoint(lines)*. This function will perform two steps (figure **??**):

1. Find, store and assign weights to the points where the lines intersect. Lines that are either too vertical or too horizintal will not be included in the calculations. Intersectionpoints outside the image frame are rejected.
2. Find the vanishing point based on the valid weighted intersection points. This is done by a voting scheme described in [GNL12], and illustrated as a flowchart in figure 3.3.

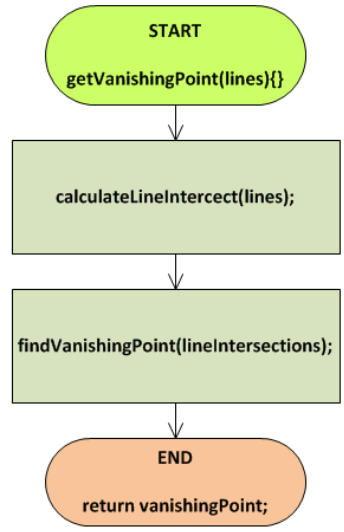


Figure 3.2: Two steps in *getVanishingPoint()*.

3.2.5 Vanishing Point Detector Application

Program Structure The vanishing point detector was implemented as a QWidget Application in the Qt Creator IDE. The code excerpt shown in algorithm 3.1 contains the most important image processing steps. A main thread, which may be called the GUI thread, handles all user related input and output. All image processing is done in the class *ImageProcessing* which inherits from QThread. This means that *ImageProcessing* controls a protected function *run()* which contains the threaded code and image processing steps. Figure 3.4 is a sequence diagram that shows the different classes and threads interact. The ellipse filter step is ommitted in the illustration; if it had been included, it would be called between *hough->detect()* and *getVanishingPoint()*.

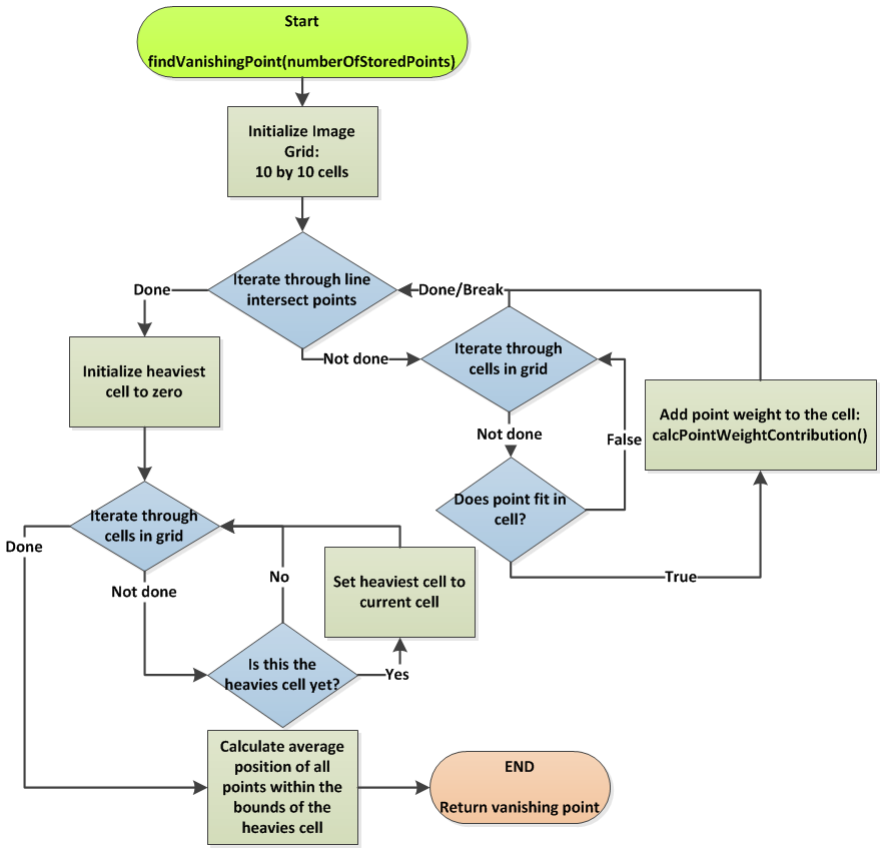


Figure 3.3: Sequence diagram illustrating program execution when the user activates camera feed and VP detection.

Algorithm 3.1 Vanishing point detector loop. Several lines of code are omitted in this example to make the processing more clear.

```
while(){
    capture.read(cameraImg);
    cvtColor(cameraImg,grayImg,CV_RGB2GRAY);
    blur( grayImg, blurredImg, Size(3,3) );
    Canny(blurredImg, edgesImg, lowerThresh, upperThresh, 3);
    gpu_edgesImg.upload(edgesImg);
    lines = detectHoughLines(gpu_edgesImg);
    newLines = mLineEllipseFilter.filterLines(lines,originalImage);
    Point vanishingPoint = vpDetector.getVp(newLines,cameraImg);
}
```

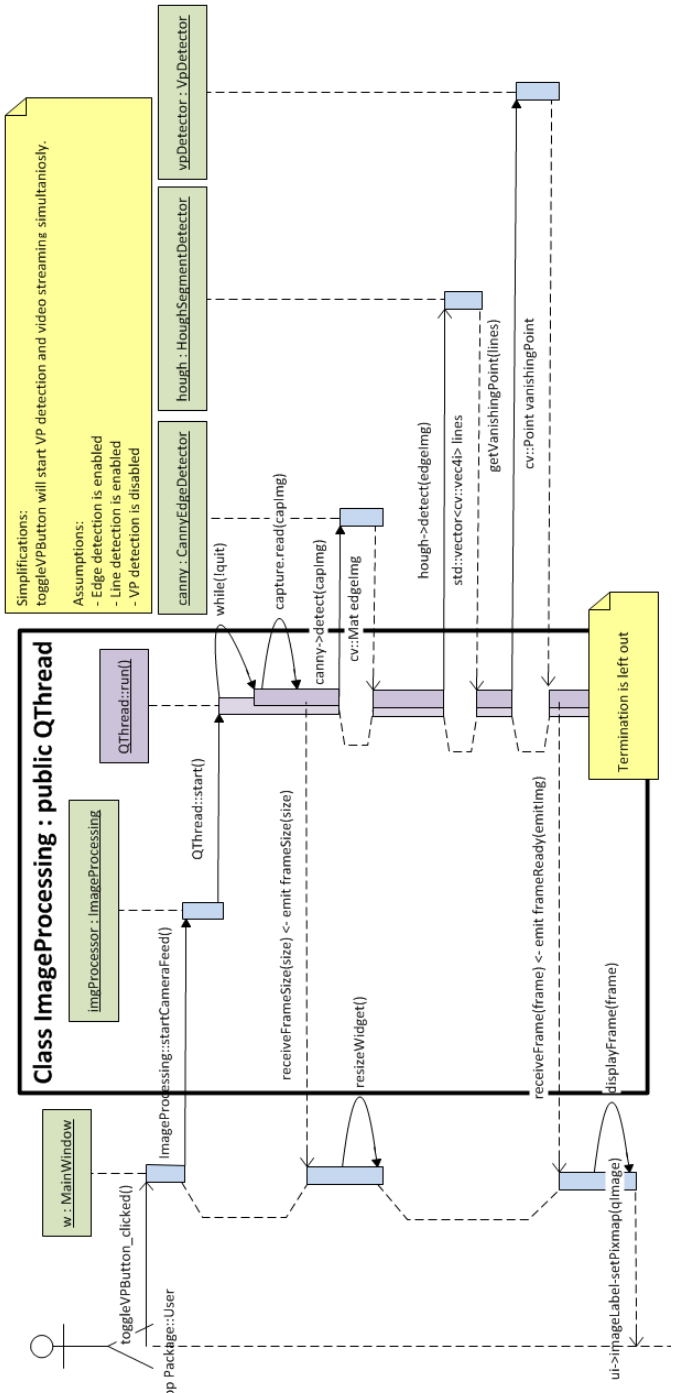


Figure 3.4: Sequence diagram illustrating program execution when the user activates camera feed and VP detection.

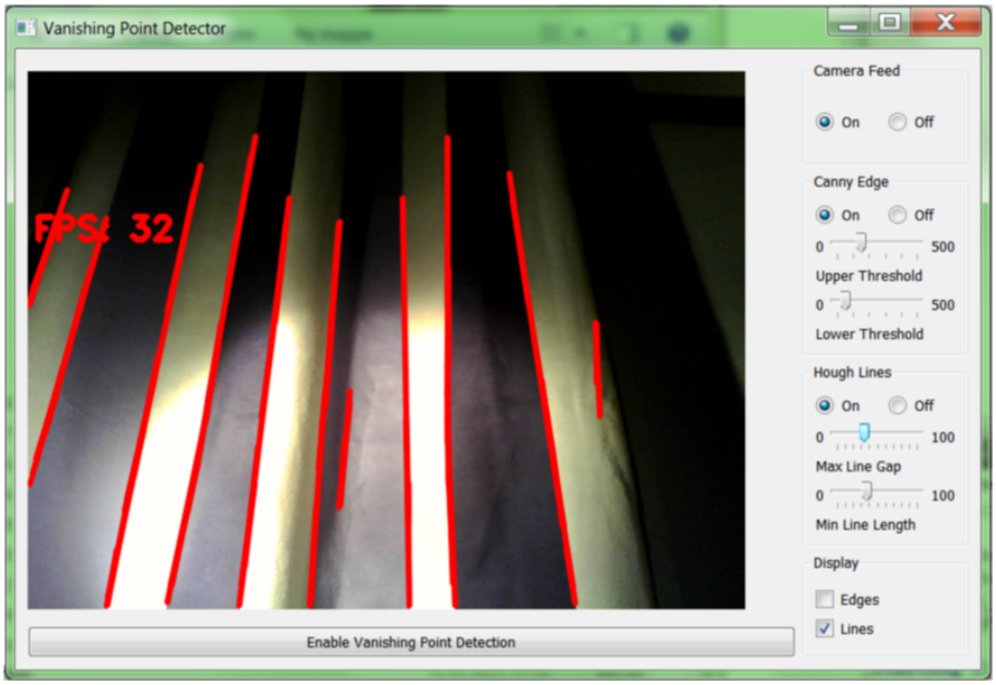
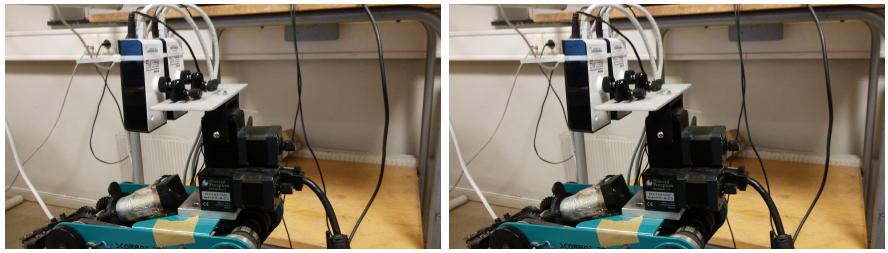


Figure 3.5: Graphical user interface for the vanishing point detector. Note that the detected line segments are actually several lines overlapping each other.

Graphical User Interface A graphical user interface was created so that the parameters for the canny edge detector and Hough lines detector could be tuned on-line. All widgets shown in figure 3.5 have their functionality implemented. The user can turn on the camera feed, in this case from the web camera integrated into the laptop of the author, and switch line and edge detection on or off. Upper and lower edge detection thresholds, as well as line detection parameters can be set by using the sliders. The kernel size for the edge detector is set to 3 by 3, and can not be changed by the user. When both edge detection and line detection is enabled, the user may turn on the vanishing point detector module. In this particular application, the ellipse line filter module is not included.



(a) Camera pair mounted on the pan-tilt module.

(b) Camera pair mounted on the pan-tilt module.

Figure 3.6: The two camera positions.

3.2.6 Cause of Failure

3.3 Depth Perception and Obstruction Detection

3.3.1 Overview

3.3.2 The camera rig

The two IP cameras were moved together to form a stereo camera. This stereo camera was used in two positions. The first camera position is on the pan-tilt module on the robot arm, see figure 3.6a. The second position is just over the LIDAR in front of the robot arm base, see figure 3.6b. The workshop at Institutt for teknisk kybernetikk (ITK) made a mounting bracket, so that the cameras could be placed over the LIDAR. In stereo vision, it is essential that the positions of the cameras relative to each other is constant. One problem encountered throughout the project was that the camera assemby, when placed either at the pan-tilt module and over the LIDAR, was not rigid enough. The severity of this problem was somewhat alleviated by wrapping a strap around both the cameras. This camera rig is ad hoc, i.e. suitable for the purpose of this project, and a better solution should be used for succeeding projects.

3.3.3 Graphical User Interface

Tuning the parameters for stereo matching in OpenCV is a wearisome task, especially without a good graphical user interface. Figure 3.7 shows the user interface which was used to observe how parameter tuning alters the disparity map quality. Not all functionalities were implemented.

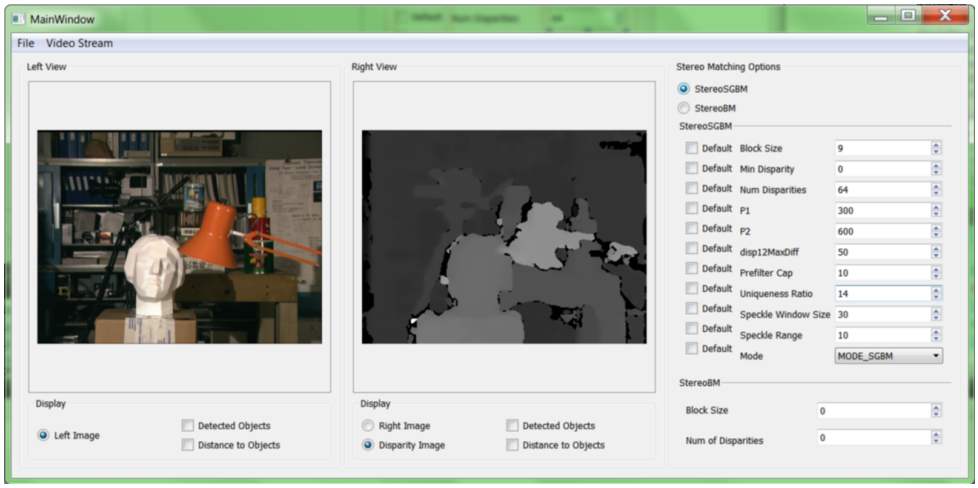


Figure 3.7: Graphical user interface for stereo matching. A disparity map is computed from the Tsukuba samples by using StereoSGBM.

3.3.4 Calibration

As mentioned in chapter 2, all cameras will have some distortion. If the distortion is too severe, as it often will be in the context of stereo vision, the camera must be calibrated. In addition, it was assumed that the image planes were located on the same plane, and that a projection pair, for example the projections $\boldsymbol{X_L}$ and $\boldsymbol{X_R}$ of an object $\boldsymbol{X}$, form two equal epipolar lines, $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$, on the two image planes. In practice, these conditions are achieved through stereo calibration. The second purpose of the calibration procedure is to relate the sensor data to real world quantities, in order to measure the distance to detected objects. Code listings from Practical OpenCV by Samarth Brahmbhatt [Bra13] has been used as a basis for calibration in this project. Some parts of his code is almost unchanged, while other parts of the listings are altered and expanded significantly. There are three steps in the calibration procedure:

1. Single camera calibration:
2. Stereo calibration.
3. Image rectification.

See figure 3.8 for an overview of the calibration procedure. All these steps require a familiar object with known dimensions to calibrate against. Among the three calibration patterns supported by OpenCV, this implementation utilized a black and white chessboard. The chessboard has 6 by 8 squares with sides $\approx \boldsymbol{26\,mm}$ long.

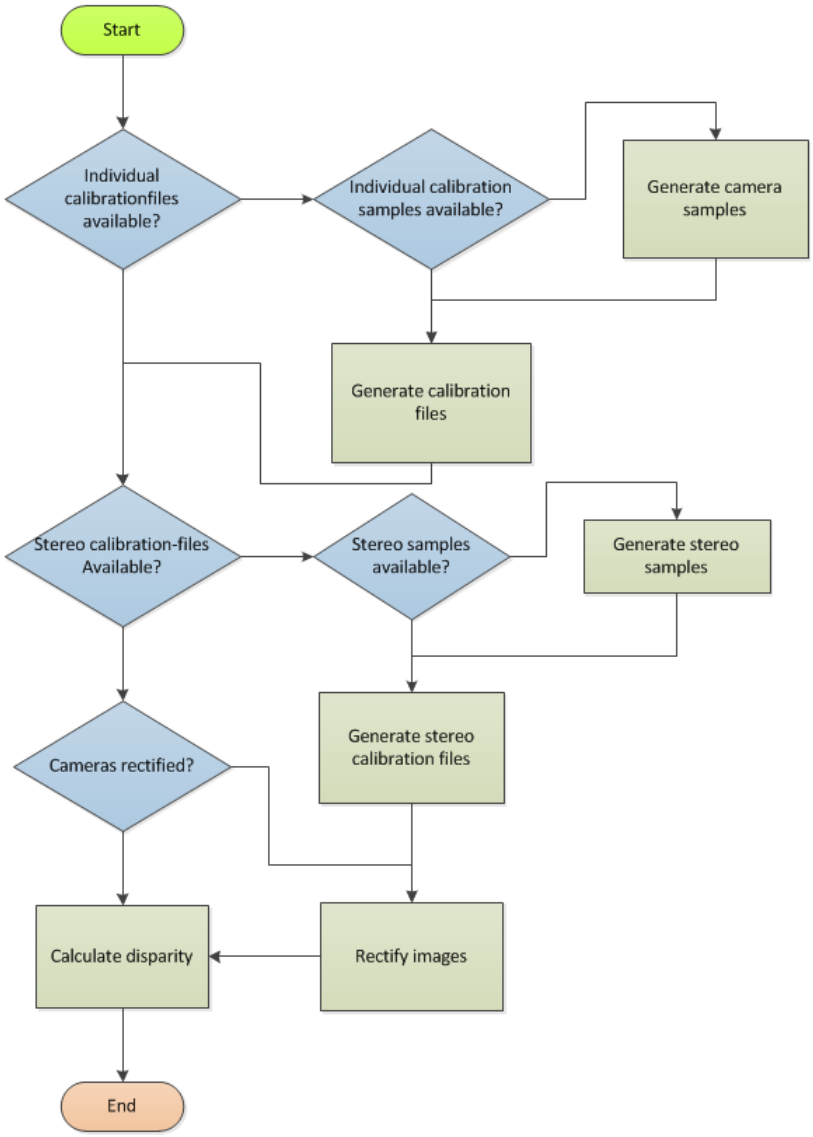
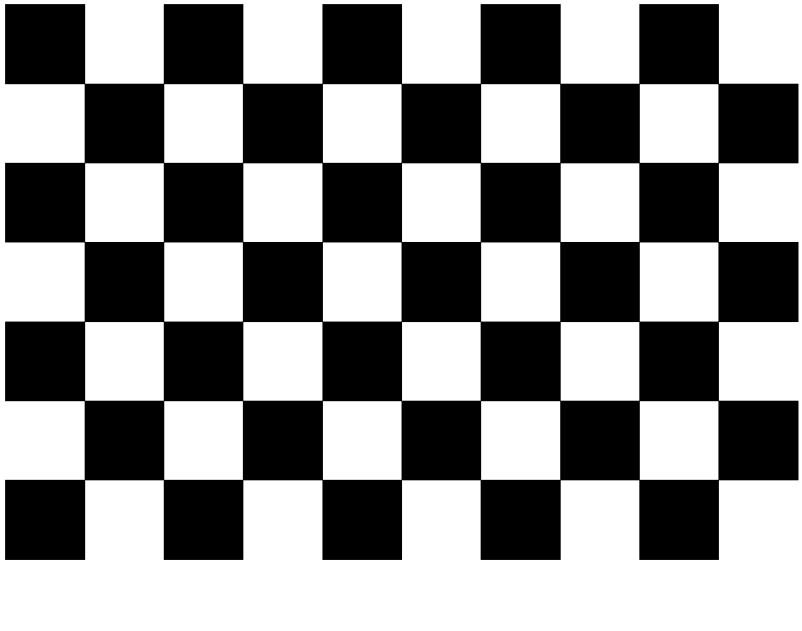


Figure 3.8: An overview of the calibration procedure.



(a) The chessboard calibration pattern. The pattern was printed and taped to a flat surface.

(b) Chessboard detection. This is one of the sample images used to calibrate the camera. Notice the distortion in the lower right corner.

Figure 3.9

Single Camera Calibration In this step, the cameras are calibrated separately. The purpose of this calibration procedure is to counter the constant radial and tangential distortion in a pinhole camera, and to identify the pinole camera model. The result of this procedure is the 3 by 3 camera matrix and the five distortion coefficients mentioned in the theory chapter. In the flowchart in figure 3.10, assume that enough useful sample images are gathered and read into the program. In the last step, OpenCVs calibration function will take inn the detected corners from the sample images and the chessboard dimentions. Finally, the calibration results are stored in an .xml file:

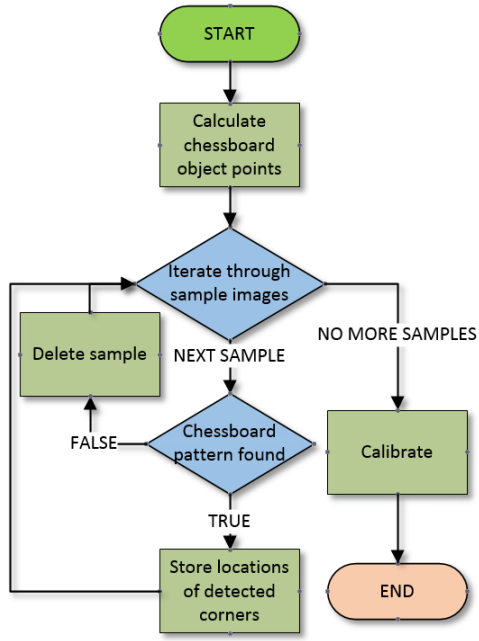


Figure 3.10: Single camera calibration.

```
<?xml version="1.0"?>
<opencv_storage>
<cameraMatrix type_id="opencv-matrix">
  <rows>3</rows>
  <cols>3</cols>
  <dt>d</dt>
  <data>
    1.4478141049219482e+003 0. 6.6274484776761142e+002
    0. 1.4432743079138295e+003 4.7609546427843065e+002
    0. 0. 1.
  </data></cameraMatrix>
<distCoeffs type_id="opencv-matrix">
  <rows>1</rows>
  <cols>5</cols>
  <dt>d</dt>
  <data>
    -2.6128696949919589e-001 3.4600669963821584e-001
    -2.2331413545278616e-003 -2.5710895791919218e-003
    -3.7144316064113458e-001</data></distCoeffs>
</opencv_storage>
```

When all the image samples has been
read into the program, the program will check if the chessboard can be detected. If the chessboard is present in the image, the position of the corners will be stored in

Stereo Calibration Stereo calibration is almost exactly the same as single camera calibration. An important difference is obviously that sample images must be taken from both cameras simultaniously. The calibration pattern must be detectable in both frames. Stereo calibration will generate a new set of data:

R: 3 by 3 rotation matrix between the two cameras.

T: Translation between the two cameras. Denoted $\boldsymbol{B}$ in the theory chapter. Relates the left camera to the right together with $\boldsymbol{R}$.

E: 3 by 3 essential matrix.

F: 3 by 3 fundamental matrix.

These matrices will be stored in a new file, ”stereo_calib.xml”, together with the distortion coefficients and camera matrices from the two single camera calibrations. These values will be applied in the next calibration step.

Stereo Rectification Rectification is the final step before stereo matching can be performed. In this step, the information that is necessary to align the stereo frames. When the frames are aligned, Rectification is completed by:

1. Loading ”stereo_calib.xml” and reading in the camera matrices, distortion coefficients, $\boldsymbol{R}$ and $\boldsymbol{T}$.
2. Calling *stereoRectify()* with the data above as input.

stereoRectify() will write data onto a set of new matrices:

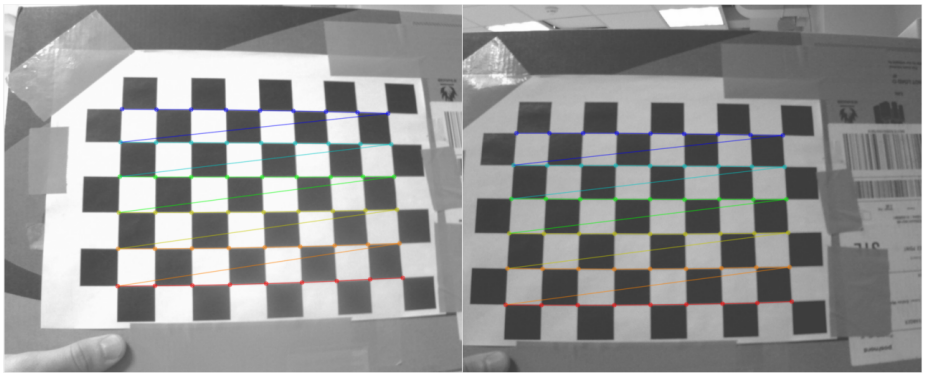
Rl: 3 by 3 rotation matrix for the left frame.

Rr: 3 by 3 rotation matrix for the right frame.

Pl: 3 by 4 left projection matrix.

Pr: 3 by 4 right projection matrix.

Q: 4 by 4 reprojection matrix. Maps disparity to depth.



(a) Before calibration and rectification.

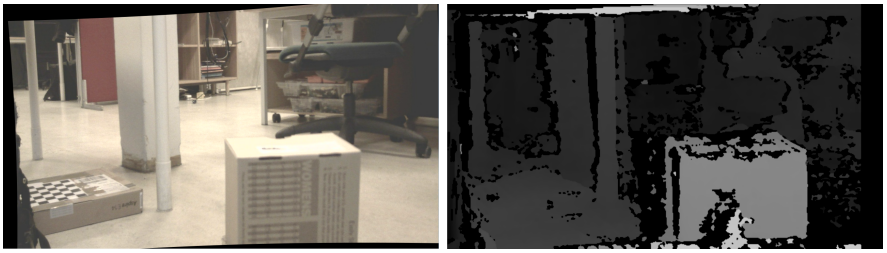
(b) After calibration and rectifivation.

Figure 3.11: Before and after calibration. The red lines in 3.11b can be used to assess the quality of the calibration procedure. Note that these image pairs are captured at different times.

Finally, the function *initUndistortRectifyMap()* will use these matrices to generate four pixel maps with a size that equals the input images. The pixel maps are added the ”stereo_calib.xml”. These matrices maps the pixels in the original images to their rectified locations. To rectify an input image, call the function *remap()*.

```
remap(leftCameraFeed, leftCameraFeed_rect, map_l1, map_l2, INTER_LINEAR);
```

In this example, the function takes in the left camera feed and the *cv::Mat* that will hold the resulting rectified image. *map_l1* and *map_l2* are the pixel maps. The result of all the calibration steps and the final rectification step is shown in figure 3.11.



(a) Left camera image. (b) Disparity map.

Figure 3.12: The result of StereoSGBM.

3.3.5 Stereo Matching

Recall that stereo matching is the search for mutual information in the sereo image pair. Matching of the sterao pair will result in a single channel disparity image (figure 3.12), i.e. the pixels have intencity values along a single dimention. The disparity is a measure of how far a block of features has shifted between the left and right image.

Processing Time In this application, with no hardware acceleration from CUDA and the like, the choice of matching algorithm stands between the regular StereoBM and the more robust StereoSGBM. Block Matching (BM) is the fastest of the two, and usually allows a frame rate of several frames per second. Semi Global Block Matching (SGBM) on the other hand, is much slower. The Tsukuba samples in figure 3.7 are 384 pixels wide by 288 pixels high. Matching them with StereoSGBM takes roughly half a second. Images from the IP cameras on the robot are set to be 1280 pixels wide by 1024 pixels high. In this case, matching with StereoSGBM takes approximatly 8 seconds; a processing time which is too long if it is to be used for navigation. A solution to this problem is to reduce the size of the image pair. This was done in the matching application by downsampling the input images to ***1/4th*** of their original size by using *pyrDown()* in OpenCV. This brings the frame rate up to 2 frames per second.

The StereoSGBM object require at least these parameters:

numDisparities: The number of disparity values, and indirectly the maximum left to right projection displacement. Must be divisible by 16.

minDisparity: The lowest disparity to be computed.

blockSize: A block of pixels. Defines the size of regions to be matched. Must be an odd number, because the block is centered around a pixel.

(a) Block size = **1**. (b) Block size = **9**. (c) Block size = **25**.

Figure 3.13: Comparison of different block sizes.

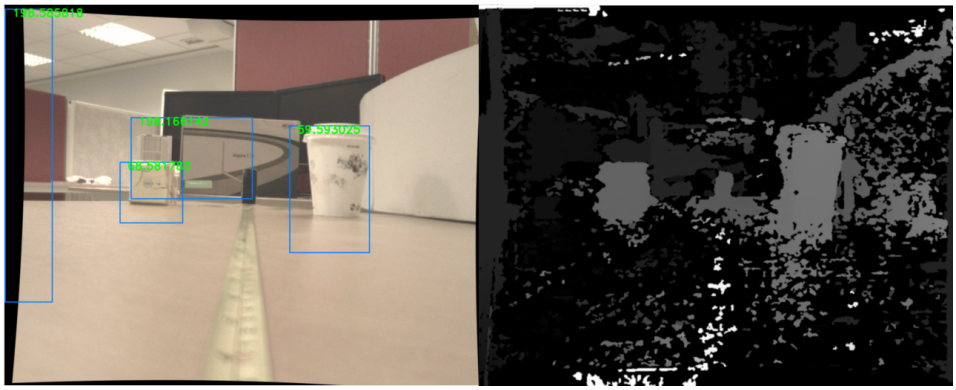


Figure 3.14: A preliminary version of the obstruction detector

Selecting a block size is a compromize between robustness and level of detail. Consider a block with the size of a single pixel. It is highly probable that it will cause false matches. A too large block will generate less noisy matches, while the details are surpressed.

3.3.6 Finding Obstructions

Detection of obstructions is based purely on color thresholding, and is performed in the custom made: *DepthFilter* class. The idea and initial code for obstruction detection is based on this autor's very first meeting with OpenCV: an object tracking tutorial by Kyle Hounslow on youtube.com. In the implemented application, obstruction detection is done within the *DepthFilter* class. After the disparity map is calculated, the disparity thread will call *exctractObstructions()* within the *DepthFilter* class. The program flow within the function is shown in figure 3.15. All detected obstructions are indicated on the original left frame by bounding rectangles (figure 3.14). The most sentral parts of the flowchart inf figure 3.15 will be explained in detail.

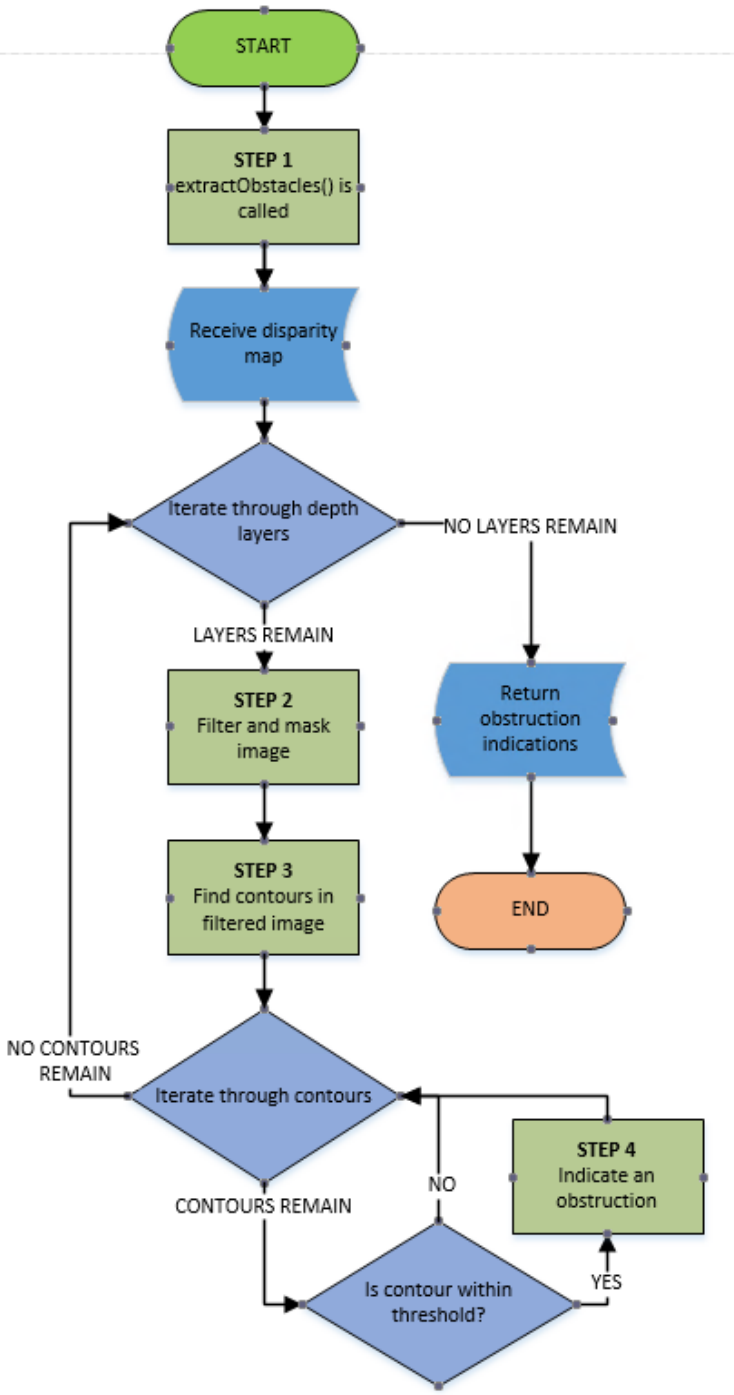


Figure 3.15: Obstruction detection in *DepthFilter*.

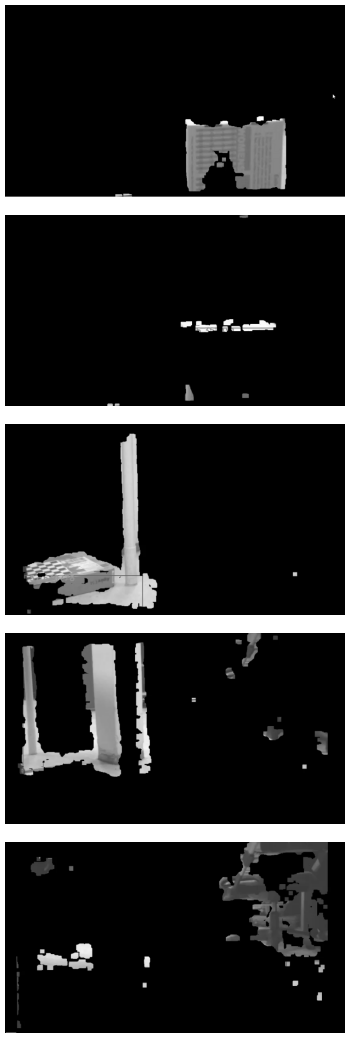


Figure 3.16: Some of the depth layers in figure 3.12 separated by color filtering. The top image is the closest layer, while the most distant layer is at the bottom.

Iterate Through Depth Layers Disparity range is set to **{0...160}**, where each value relates to a corresponding distance. This part of the code will iterate through disparity intervals with a constant size of 10 starting at an interval of **{150...160}**, and ending at **{20...30}**. The idea behind these intervals is that possible obstructions will stand out from the surface of the ground level (floor) and form a distinct shape that can be separated from the surroundings.

Filter and Mask Image This step will produce a binary image which represents the region of the disparity image where the disparity values falls within the interval from the previous step. This binary image is filtered by eroding all features with a size smaller than a predetermined value.

Find Contours It is possible to detect a set of contours in the filtered binary image. This is done with a prebuilt function from the OpenCV library.

Iterate Through Contours and Indicate Obsticles Some of the contours are so small that they can either be considered to be noise or harmless obsticles. Contours with an area above a certain size are also rejected. Rejecting large contours may seem to be a bad decision, but is is necessary given the limitations of this implementaton4. The contours are then indicated by a bounding rectangle.

3.3.7 Distance Measurment

Another functionality implemented within the *DepthFilter* class is the ability to calculate the distance from the camera to the detected contours.

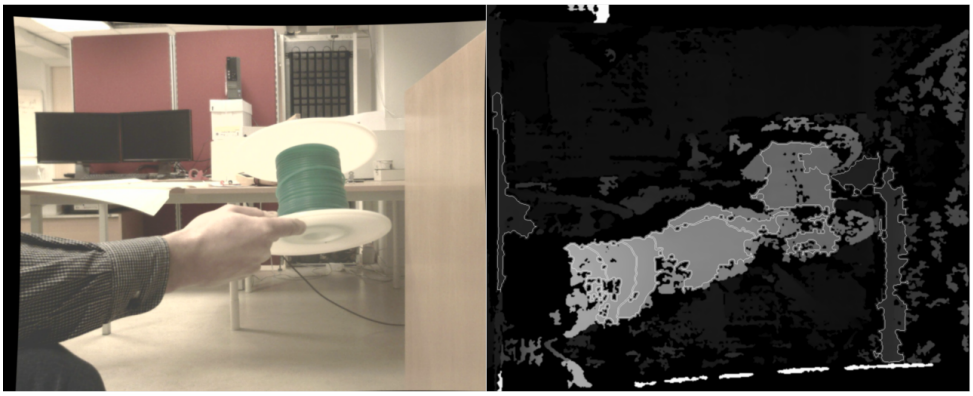


Figure 3.17: All the detected contours in a disparity map that falls within the size threshold.

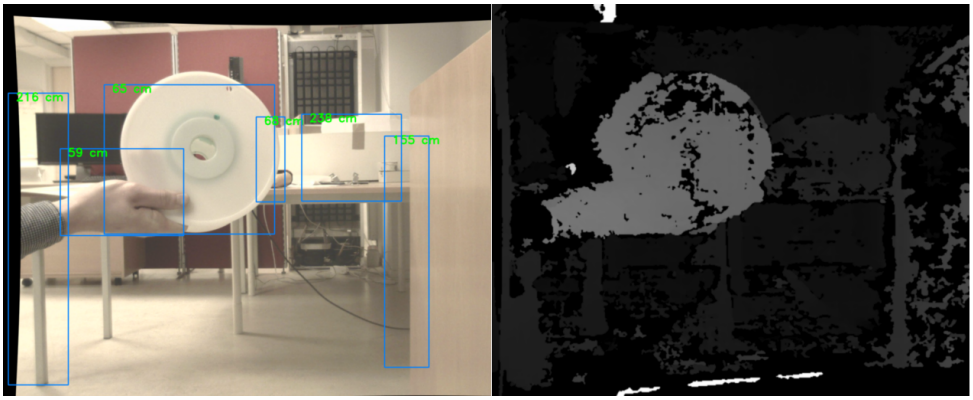
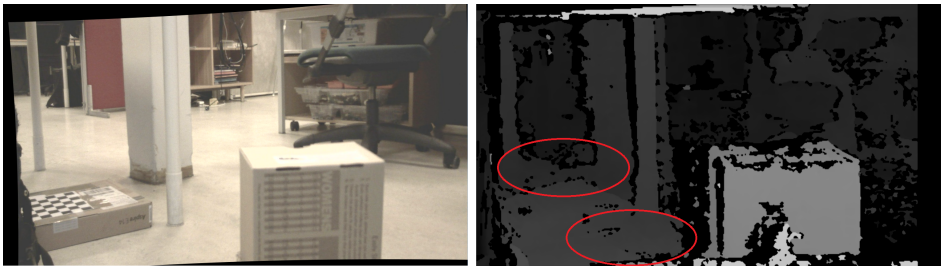


Figure 3.19: Obstruction detection in *DepthFilter*.



(a) Left camera image. (b) Disparity map.

Figure 3.20: The floor will have the same disparity value as two distict obstructions, thus making them appear as a single obstruction.

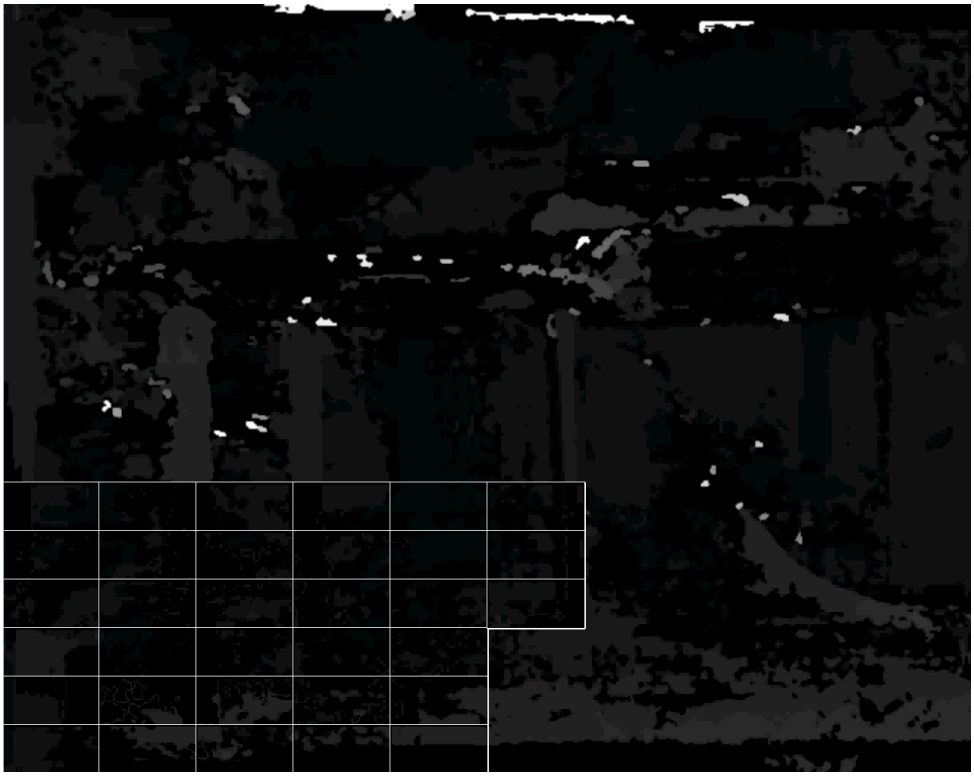


Figure 3.21: Floor filtering in progress.

3.3.8 Floor Filtering

Figure 3.18: A mask.

Chapter 4
Assessment

4.1 Introduction

This chapter will discuss the implementatons in the previous chapter. Intended functionality and performance is compared to the actual results, and usefullness in the context of autonomous navigation is assessed. The assesments will be mostly qualitative in nature.

4.2 Vanishing Point Detection

The vanishing point detector failed work as intended. Both the DSaM algorithm and the vanishing point detector itself is unable to fulfill their intended tasks.

4.3 What Does Work?

The application is stable and responsive as long as the DSaM algorithm is left out. While eliminating all errors and bugs was too time consuming to be completed, testing showed that the vanishing point voting scheme is working properly. It is suspected that the error is caused by mixing two coordinate systems.

4.4 Depth Perception and Obstruction Detection

4.4.1 Stereo Vision

The stereo matching process has two significant weaknesses: inability to detect textureless surfaces, and false disparities from repetitive patterns. It may be that the matching parameters can be tuned in order to achieve denser disparity maps. On the other hand, the input images from the IP cameras are quite noisy, which makes the matching process much more difficult. It is possible that areas of the images where the distictive features are faint are especially sensitive to noise. This

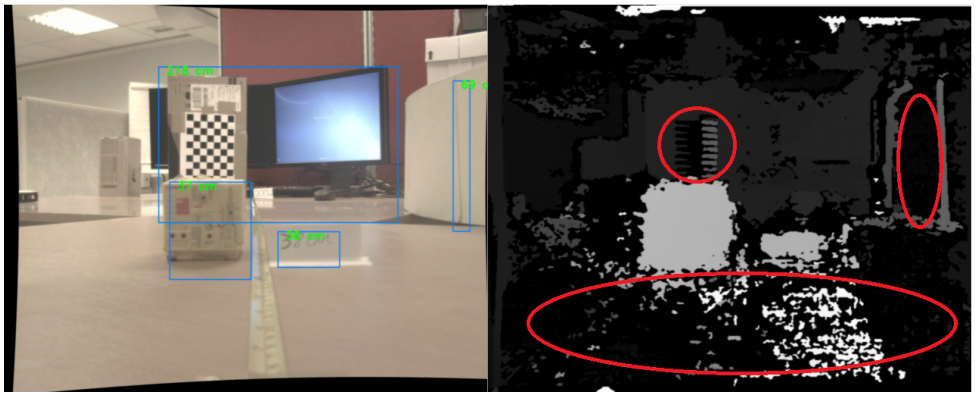


Figure 4.1: Illustration of the weaknesses with stereo matching. Little or no matching where there are few distict features. False matches and shadows caused by reperitive patterns.

results in a sparse disparity map, which can be unsuitable in applications such as 3d reconstruction and mapping of the environment.

4.4.2 Obstacle Detection

Obstacle detection based on depth layers works as intended, and could potentially be adapted and improved for an obstacle avoidance system. Figure 3.17 in chapter 3 shows that several contours are placed closed to each other, thus resulting in an image cluttered with detected obstructions. While a cluttered image is not user friendly, it is not necessarily a problem in terms of obstacle detection. Note that this is an obstacle detector, not an object detector. This implies that it is sufficient to know that something is obstructing the path of the robot.

Chapter 5
Conclusion

5.1 Future Work

Of the two implementations, the obstruction detector appears to be the better candidate for further developement.

5.1.1 Integration With Point Cloud Library (PCL)

A great deal of time was spent on an attemt to integrate Qt and OpenCV with the Point Cloud Library (PCL). Needless to say, the integation was not successful. PCL is an open source procject for image and point cloud processing [**?**]. Similarily to OpenCV, it is free of charge, and the source code is available for download on the project home page and on GitHub. PCL depends on many 3rd party libraries which must be downloaded and compiled separatly. This complicated the integration process, and was the most significant hindrance to a successful integration.

5.1.2 New Hardware

Visual Sensors The current camera set-up comes with limitations that makes them unsuitable as navigational sensors. The video feed is unsyncronized, which makes the disparity map useless whenever there is relative motion between the cameras and the surroundings.

Computing Hardware Suitable for Image Processing

5.2 Task Fulfilment

5.3 Final Conclusion

References

[3dC] Camera calibration with opencv. http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html.

[Asp13] Petter Aspunvik. Robotisert vedlikehold. Master's thesis, NTNU, 2013.

[Ber13] Mikael Berg. Navigation with simultaneous localization and mapping. Master's thesis, NTNU, 2013.

[Bra08] Gary Bradski. *Learning OpenCV*. O'Reilly Media, 2008.

[Bra13] Samarth Brahmbhatt. *Practical OpenCV*. Apress: Berkeley, CA, 2013.

[can] Canny edge detection with opencv. http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html.

[DRMS07] A.J. Davison, I.D Reid, N.D Molton, and O. Stasse. Monoslam: Real-time single camera slam. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6), jun 2007.

[DWB06] H Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *Robotics & Automation Magazine, IEEE*, (2), jun 2006.

[GNL11] Demetrios Gerogiannis, Christophoros Nikou, and Aristidis Likas. A split-and-merge framework for 2d shape summarization. *Image and Signal Processing and Analysis (ISPA), 2011 7th International Symposium on*, pages 206 – 211, sep 2011.

[GNL12] Demetrios Gerogiannis, Christophoros Nikou, and Aristidis Likas. Fast and efficient vanishing point detection in indoor images. *Pattern Recognition (ICPR), 2012 21st International Conference on*, pages 3244 – 2347, nov 2012.

[Hir08] Heiko Hirschmuller. Stereo processing by semiglobal matching and mutual information. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(2), 2008.

[LC12] Chao Liu and L. Christopher. Depth map estimation from motion for 2d to 3d conversion. In *Electro/Information Technology (EIT), 2012 IEEE International Conference on*, pages 1–4, May 2012.

[NN00] Firstname 1 Name1 and Firstname2 Name2. A dummy title. *A Fake Journal*, 1(1):000–000, June 2000.

[Pra80] K. Prazdny. Egomotion and relative depth map from optical flow. *Biological Cybernetics*, 36(2):87–102, 1980.

[XWS06] Jian Xu, Han Wang, and Andrew Shacklock. Visual guidance for autonomous vehicles. In *Autonomous Mobile Robots*, chapter 1, pages 5 – 40. CRC Press, may 2006.

Appendix A

Setting up a project with Qt and OpenCV