NTNU

Norwegian University of
Science and Technology

Perception for obstacle-aided locomotion of snake robots

Fredrik Leine Lerø

Master of Science in Cybernetics and Robotics
Submission date: June 2017
Supervisor: Øyvind Stavdahl, ITK
Co-supervisor: Esten Ingar Grøtli, Sintef
Jon Azpiazu, Sintef

Norwegian University of Science and Technology
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Faculty of Information technology,
Mathematics and Electrical Engineering
Department of Engineering Cybernetics

Master's Thesis

Student's name: Fredrik Leine Lerø

Field: Engineering Cybernetics

Title (Norwegian): Omgivelsesbeskrivelse for slangeroboter

Title (English): Environment representation for a snake robot

Description:
In nature, snakes are capable of performing an astounding variety of tasks. They can locomote, swim, climb and even glide through the air in some species. However, one of the most interesting features of biological snakes is their ability to exploit roughness in the terrain for locomotion, which allows them to be remarkably adaptable to different types of environments. To achieve this adaptability and to locomote more efficiently, biological snakes may push against rocks, stones, branches, obstacles, or other environment irregularities. They can also exploit the walls and surfaces of narrow passages or pipes for locomotion.

The term perception-driven obstacle-aided locomotion may be adopted to define locomotion where the snake robot utilizes a sensory-perceptual system to exploit the surrounding operational space and identifies walls, obstacles or other external objects, for means of propulsion. SINTEF and NTNU have already produced some initial results within this area and have ongoing projects to further develop snake robot capabilities.

The aim of this project is to use Computer Vision based methods to generate a representation of the environment, and process such representation in a way that the output can be used by the snake robot's control algorithms to plan a trajectory. The problem can be formulated in terms of a Visual SLAM approach, in the sense that the snake robot does not have any previous information of the environment, and must generate such representation as it moves through the space, while simultaneously localizing itself. One of the key aspects in comparison with commonly used methods is that the representation needs to be dense - as it is not only used for localization, but also for path planning. The tasks to be carried out are:

1. Study existing methods for dense or semi-dense Visual SLAM
2. Setup dense or semi-dense Visual SLAM algorithms using monocular vision or RGB-D camera
3. Propose extensions to existing methods that account for the requirements of the snake robot application - specially the need for a full dense representation
4. Implement some of the proposed extensions and integrate into the SLAM pipeline
5. Perform an exhaustive experimental evaluation
6. Propose (and potentially implement) an architecture for integration with the snake robot's control system

Advisor(s): Øyvind Stavdahl, Assoc. Professor, Dept. of Engineering Cybernetics, NTNU
Esten Ingar Grøtli, , PhD, Control System Engineering, SINTEF
Jon Azpiazu

Trondheim, <19.06.17>

Øyvind Stavdahl
Supervisor

Summary

The goal of this project was to provide an obstacle-aided snake robot with improved perceptual capability. This implies ensuring that objects located in the snakes' close proximity get a high resolution digital representation. Kintinuous, an algorithm used for mapping an unknown environment while at the same time keeping track of an agents location within it, is the perceptual system of the snake robot. The algorithm maps the environment by getting RGB-images and depth data from a depth sensor called Kinect. The Kinect has a limited range for where it can obtain depth data. Objects located outside this limited range will get a low resolution digital representation, or no representation at all. To compensate for this limited range, Kintinuous was altered by adding a new system component responsible for computing depth maps with high resolution at lower ranges. The depth maps were computed by using stereo vision principles. The Kinect has a single RGB-camera that is used to capture two different views of the scene, at different times. The new system component then proceeds to compute the matrix relating the two different views in pixel coordinates, by exploiting the Kintinuous tracking information. The images are then rectified to make them row aligned, a dense disparity map computed by matching corresponding pixels, and a depth map acquired. The results of each part of the system showed the rectification to be inaccurate, resulting bad performance of the matching algorithm. This lead to depth maps being very noisy or completely useless. The depth map fusing was with Kintinuous was though never achieved. Further work is necessary to make this work as intended, and this thesis is believed to form a good basis for the development of a capable algorithm.

Sammendrag

Målet med denne oppgaven var å skaffe en slagerobot som utnytter hindringer for å skyve seg avgårde forbedret digitalt syn. Dette innebærer å forsikre seg om at gjenstander som befinner seg i nærheten av slangeroboten får en høyoppløslig digital representasjon. Slangens digitale syn er gitt av algoritmen Kintinuous. Kintinuous er i stand til å lage et kart av et ukjent området, samtidig som den er i stand til å vite hvor slangen beginner seg i det. Kintinuous er i stand til å kartlegge området ved å få fargebilder og dybde data fra en sensor som heter Kinect. Kinect sensoren har en begrensning for hvor den kan få dybdedata. Objekter og hindringer som ligger utenfor dette området får en lav oppløst digital representasjon, eller ingen representasjon i det hele tatt. For å komponsere for denne begrensningen, Kintinuous var utvidet ved å legge til en ny system komponent ansvarlig for å lage høyoppløslige dybdekart når objekter befinner seg utenfor begrensningen. Dybde kartene ble laget ved å bruke prinsipper fra stereo vision. Kinect sensoren har ett RGB-kamera som brukes for å ta to bilder av omgivelsene fra forskjellige vinkler, tatt ved ulike tider. The nye systemkompnenten vil deretter regne ut en matrise som relaterer de forskjellige vinklene i piksel kordinater, ved å utnytte tracking dataene som kommer fra Kintinuous. Bildene er deretter forvridd for å sørge for at motsvarende punkter ligger på samme bilderad, en prosess som kalles rectification, før et tett dybdekart er laget ved å finne den motsvarerende piksler for alle pikslene i bildene. Resultatene viste at ”rectification” prossesen var unøyaktig, noe som førte til dårlig ytelse av matchings algoritmen. Dette første til at dybdekartene var veldig støyete, eller helt ubrukelig. Å jobbe videre med dette er nødvendig for at systemet skal fungere som tiltenkt.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1 Background and Motivation

Biological snakes are extremely efficient in their ability to exploit obstacles in their environment for locomotion[6]. They may push against rocks, branches, and other environmental irregularities to propel themselves forward. There is ongoing research into the possibility for snake robots to adapt this ability of exploiting the environment for more efficient locomotion. Initial results have already been produced within this area, and it is still being researched[7]. The acquisition of obstacles suited for locomotion can be achieved through the use of perceptual sensory. This can be defined as *perception-driven obstacle-aided locomotion*[8]. This thesis is complementary to [9], which explored the possibility of using Kintinuous[5], an extension to the KinectFusion[10] algorithm, as the snake robots Simultaneous Localization and Mapping (SLAM)[11] system. KinectFusion is an algorithm developed by Microsoft that is able to create a highly detailed 3D model of the scene in real time. It depends on getting RGB-images and depth maps as input, which is provided by a 3D sensor, also developed by Microsoft, called the ”Kinect”. However, among other limitations, KinectFusion's mapping ability is bounded to a small region around the physical point of its initialization. Kintinuous extends these capabilities to permit KinectFusion style mapping in an unbounded environment. Kintinuous is thus able to map the geometry of an unbounded, and unknown, environment while keeping track of where the sensor currently is, and previously has been, located. In the exploration of using Kintinuous as a snake robots SLAM system, obstacles potentially suited for propulsion was located by segmenting the environment generated by Kintinuous into smaller clusters. Each individual cluster was then evaluated with respect to size and shape. The initial results of this project showed the method to be suited for testing in an ideal environment known beforehand, with the sensor operating within its optimal range. However, during a realistic scenario, obstacles may, more often than not, be located at a close proximity to

the snake, and consequently lie outside the Kinect sensors operating range. For Kintinuous to be considered a valid option for the snake robots SLAM system, the generated environment should include high resolution depth measurements of objects located close to the sensor. This thesis explores the possibility of compensating for this limited range by the use of stereo vision principles, and exploiting the tracking information provided by Kintinuous.

In recent years, the concept of monocular SLAM has been well studied and spawned two main approaches to solving the SLAM problem. Namely, feature-based and direct methods [12]. ORB-SLAM [13] is a feature-based approach that creates a sparse reconstruction of the environment, and tracks the agents location by extracting and matching strong feature-points between input images. Feature-based methods are generally more robust than direct methods, as long as the environment being mapped is feature-rich. LSD-SLAM [14] is a direct approach that builds a semi-dense reconstruction of the environment, while also tracking the location of the agent by directly operating on image intensities. Tracking is achieved by direct image alignment, while the geometry is obtained by filtering over many pixelwise stereo comparisons. This makes LDS-SLAM able to produce a much denser 3D reconstruction than its counterpart.

Stereo vision is the process of determining 3D information of the environment by matching and triangulating corresponding points between two different views of the scene. This is done by the use of a Stereo camera; two cameras mounted at a fixed distance from each other, assuring two different views of the scene. In stereo vision, the problem of getting accurate and dense 3D reconstructions by stereo matching is a challenging task. There are dozens of publications on how to deal with occlusions, object boundaries and fine structures when doing stereo matching. Stereo matching algorithms try to locate the corresponding points between two different views of the scene. The difference in horizontal coordinates between each correspondence is referred to as the *disparity*. By knowledge of the geometrical relation between the camera's, and their intrinsic parameters, depth can be determined from a disparity map by triangulation. There are two main approaches on how to solve the stereo matching problem, them being global and local methods. Disparity computation done by global methods usually try to minimize a global energy function that includes summing up pixelwise matching costs while favouring smooth disparities. Kolmogorov and Zabih's Graph Cuts Stereo Matching Algorithm [15] is a global method that incorporates graph cuts, which handles occlusions. While the quality of the computed disparity maps with this algorithm is impressive, it is at the cost of computation time. Local methods usually incorporates a "winner takes all" scheme, selecting the disparity value with the lowest matching cost. A simple, yet common, local stereo matching algorithm is called Block Matching [2]. It works by correlating image patches (called blocks) between left ant right stereo images to find the correspondences.

Improving snake robots' ability to locomote more efficiently has a great number of applications. By being adaptable to roughness in the terrain, snake robots could possibly explore dangerous areas, go on search and rescue missions, and assist firefighters[2]. This serves as a motivation for the thesis.

1.2 Goal and Method

The Kinect sensors ability to produce high quality depth maps at a frequency of 30Hz is a key component of making the Kintinuous algorithm a robust Visual SLAM system. However, the algorithm is limited by the effective range of the Kinect sensor. The sensor is unable to produce high resolution depth maps for objects located within its minimum range, that being 0.6 meters for the sensor used. The goal of this thesis is to investigate the possibility of compensating for the Kinect sensors minimal range by the use of stereo vision principles. To achieve this goal a number of smaller objectives must be performed in a specific order. To compensate for the Kinect sensor's limited range, depth maps at closer ranges must be obtained. The depth maps can be obtained by exploiting stereo vision principles, which requires a stereo matching algorithm. In turn, a stereo matching algorithm requires two different views of the scene. Due to this coherent dependency, clear objectives present themselves that must be done successively. Specifically, these objectives are:

- Extensive research into stereo vision theory.
- Analysis of the Kintinuous source code in order to locate exploitable system components.
- Exploit these system components to obtain the necessary data.
- Implement a multiview geometric reconstruction algorithm capable of producing depth maps by the use of RGB camera located on the Kinect sensor
- Fuse the depth maps with the environment representation generated by Kintinuous to improve depth resolution at close ranges.

To accomplish these objectives, research into how to use the stereo vision theory in practice is necessary. As Kintinuous is implemented by object-oriented programming in C++, getting familiar with the Open Computer Vision library is imperative.

1.3 Contribution

The novelty of the problem explored in this thesis is that it requires to fill in for the out-of-range Kinect by the use of a single RGB-camera. This involves computing depth maps, by having a moving camera to obtain at least two different views of the scene, in contrast to classical stereo vision which uses two calibrated cameras for the same purpose. The Graph Cuts [15] and Block Match [2] stereo matching algorithms mentioned earlier computes a disparity map by using a stereo camera. For both cases, the two different camera views are always related to each other in the same way. This is different from the method explored in this thesis, as obtaining two views from single camera that can move freely can results in an infinite different relations between the views. Furthermore, to compute a depth map from the two views, this geometrical relation must be known. The problem is therefore based on exploiting the tracking data provided by Kintinuous to account for the missing information. Kintinuous provides a dense reconstruction of the scene, which consequently

means that the depth maps from the developed system also must be dense for the fusion. ORB-SLAM [13] and [14] is similar with regards to that both are monocular systems. However, the environment reconstruction created is sparse for ORB-SLAM and semi-dense for LSD-SLAM. The combined problem of making a dense reconstruction of the environment by a single camera with no consistent relation between the two views used to create the depth maps has never been considered in the literature before, and a solution would therefore be considered unique.

1.4 Outline of the Report

The report is divided into four main parts. The necessary background theory is presented in chapter 2, which provides an overview of the stereo vision theory and a more in depth explanation of KinectFusion, Kintinuous and the Kinect. Chapter 3 gives an overview of the system developed as a whole, and each system component individually. Chapter 4 presents the experiments conducted and the obtained results from the thesis. Chapter 5 discusses the conduction of the goal, the results and gives recommendations for future work. Chapter 6 concludes the thesis.

CHAPTER 2

Basic Theory

This chapter aims to give the reader an overview of the concepts from computer vision and stereo imaging used in the system implementation, as well as a presentation of how Kintinuous functions. Section 2.1 provides a basic overview of digital images, image features and their descriptors. Features and descriptors are a big part of many computer vision concepts, e.g stereo imaging, which fundamentals are camera basics and projective geometry. Sections 2.2 provides the necessary theory on these fundamentals. Section 2.3 audit stereo imaging. In practice, stereo imaging involves three steps when starting of with a stereo image pair to the computation of a depth map; rectification, matching and triangulation. Section 2.3.1 starts of with triangulation to motivate the first two steps. Then section 2.3.2-2.3.3 provides the theory on Epipolar geometry and the Fundamental Matrix, concepts that directly concern rectification. Finally, rectification and stereo matching are explained in sections 2.3.4 and 2.3.5, respectively.

Section 2.4 gives an overview of depth sensor, and specifications of the actual depth sensor used in this thesis. Before auditing Kintinuous, section 2.5 provides a brief introduction to visual SLAM in general. Section 2.6 explains the system of KinectFusion, which is the algorithm Kintinuous is built upon, before finally presenting Kintinuous itself in section 2.6.

2.1 Computer Vision

2.1.1 Digital Images

A digital image [16] is a two-dimensional matrix of pixels. The resolution of the digital image is represented as the number of pixels contained by the image width, and the number of pixels contained by the image height, i.e., width $\times$ height. A pixel is the smallest controllable element in a digital image, and the size of the individual pixels will vary

among devices. To illustrate this, imagine the digital images displayed by a smartphone with the same resolution as a 50 inch TV. Depending on the resolution, it is easy to spot the individual pixels on the TV screen, in contrast to the smaller screen on the smartphone. The pixels are addressed by their physical location on the digital image, and hold a number of values depending on the color model of the image. If the image is gray scale, each pixel will hold a single intensity-value ranging from 0 (black) to 255 (white). If the image is represented by the RGB color model, each pixel will hold one intensity-value for each color channel, that is red, green and blue. Thus, a digital image taken by a camera contains three pieces of information about the scene. The intensity of the pixel representing the color of this particular spot in the scene, the image x-coordinate of the pixel and the image y-coordinate of the pixel. When humans look at a camera image, we are to some extent able to determine the depth of different objects in the scene [2]. This is purely due to our domain knowledge of the world. Computers do not have this knowledge at their disposal, and are consequently not able to determine depth from one digital image.

2.1.2 Features and Descriptors

Consider the image shown in figure 2.1. Six small image patches are given at the top of the image.

Figure 2.1 Image with extracted patches. Taken from [1]

A and B are flat surfaces that are quite difficult to deduce where they come from. C and D have edges of the building, so it is much simpler to deduce which part of the image correspond to the patches. However, it is still difficult to pinpoint the exact location where the patches come from. Lastly, E and F contain corners of the building, and as a consequence finding the exact location where the patches come from is easy. These are what can be regarded as good features in the image because they are unique, can easily

be tracked and easily compared. The Open Computer Vision (OpenCV) library, which is depended heavily on in Kintinuous and this thesis, has an arsenal of feature detecting algorithms for finding such features. The output of these algorithms are feature *Keypoints*, meaning points in the image where large variation in intensity happens in all directions. Consider the image in figure 2.1 again. When trying to find where the individual patches given come from in the image, we describe the patches in our minds, and try to find where in the image this description fits. Take patch E for example. The upper part of the image is the sky, the lower part is part of the building with some glass, etc. With this description of the feature, we are able to locate where it comes from by finding the point in the image whose neighborhood matches this description. In a similar way, computer vision descriptor algorithms describe the neighborhood of a feature keypoint so it can be found in other images. This is called *Feature Description*. OpenCV provides many methods for finding features and their descriptors[17]. There are costly variants, such as Scale-Invariant Feature Transform (SIFT) and Speeded-Up Robust Features (SURF) which takes longer to compute but are more robust than less costly methods, such as Binary Robust Independent Elementary Features (BRIEF) and Oriented FAST and rotated BRIEF (ORB). Less costly methods are much faster to compute, but are not as robust as their costly counterparts. Feature descriptors encode information about the feature neighborhood into a vector of numbers, and can be regarded as a keypoint's numerical "fingerprint". The numbers can either be real-values (SIFT) or binary (ORB). For matching descriptors, the similarity of the descriptor-vectors are measured by calculating the distance between them, e.g in Euclidean space for real numbers or the Hamming distance for binary numbers.

2.2 Camera Basics

This section covers some basic theory on cameras and projective geometry, which is essential to be familiar with before moving on to stereo imaging.

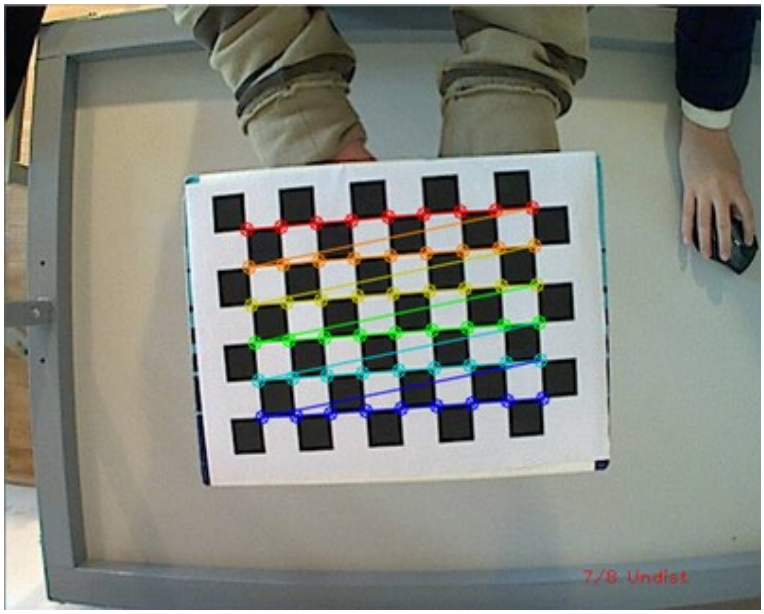
2.2.1 Camera Model

The simplest model for how a camera functions is the pinhole camera model [2]. This model consists of a wall that blocks all rays of light, except for those passing through a tiny hole in the center. In this imaginary model only a single ray enters from any particular point and gets projected onto the image plane. In real life, a pinhole does not gather enough light for rapid exposure, and that is why real cameras, and our eyes, use lenses to gather more light than what would be available at a single point. However, using a lens introduces distortion to the image. By using camera calibration one is able to correct this distortion caused by the lens, and also explain the relation between a camera's natural units (pixels) and the units of the physical world (meters). The parameters produced by camera calibration are called the intrinsic and distortion parameters. In essence, camera calibration estimates the parameters of the image sensor and the lens of a camera.

Distortion

Before further presenting the pinhole camera model, consider the image of a checkerboard in figure 2.2.

Figure 2.2 Distorted image of a checkerboard. Image taken from OpenCV documentation



The curving of the edges on the checkerboard and the edges of its surroundings is caused by distortion. Distortion comes in two forms, mainly radial and tangential distortion. Radial distortion is also known as ”barrel” or ”fish-eye” distortion, and results in straight lines being curved inwards. This is due to the field of view being ”squeezed” to fit the image plane, because the field of view of the lens is much wider than that of the plane. Tangential distortion occurs in the form of the image being tangentially displaced relative to the camera center, and is intuitively caused because the lens is not perfectly parallel to the image plane. By camera calibration, the distortion and intrinsic parameters can be obtained and used to undistort the images. Figure 2.3 shows an image of the checkerboard, now undistorted, which can be seen clearly by the straight lines.

Figure 2.3 Undistorted image of a checkerboard. Image taken from OpenCV documentation

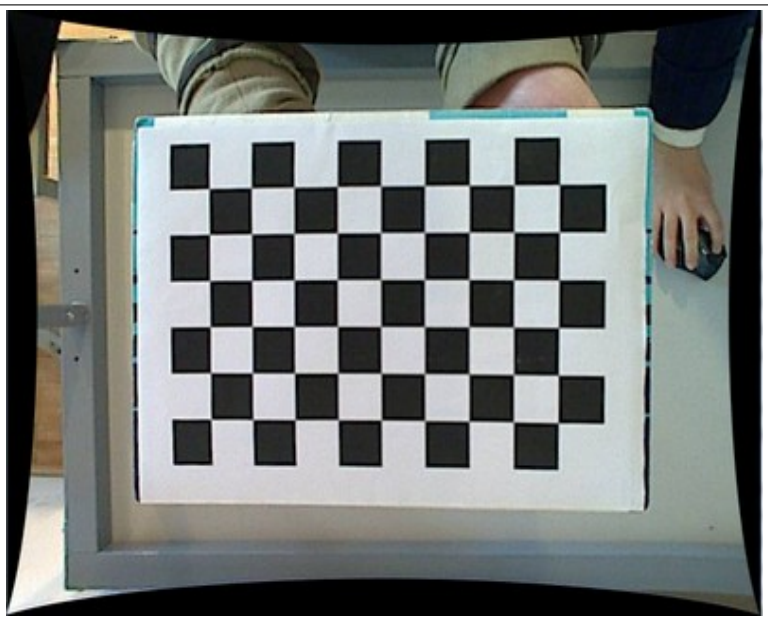


Figure 2.4 Pinhole camera. Taken from [2].

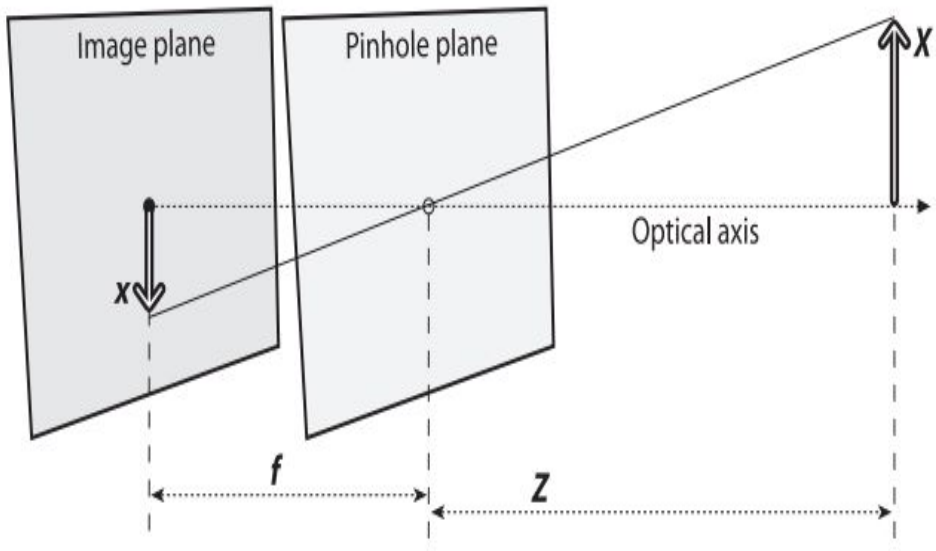


Figure 2.4 shows the ideal pinhole camera model. Only a single ray from any particular point enters the pinhole and is projected onto an imaging surface. The size of the image relative to the distant object is given by a single parameter, namely its focal length. The focal length is the distance from the pinhole aperture to the imaging screen. Two equivalent triangles form, as seen in the figure, that introduces the relationship:

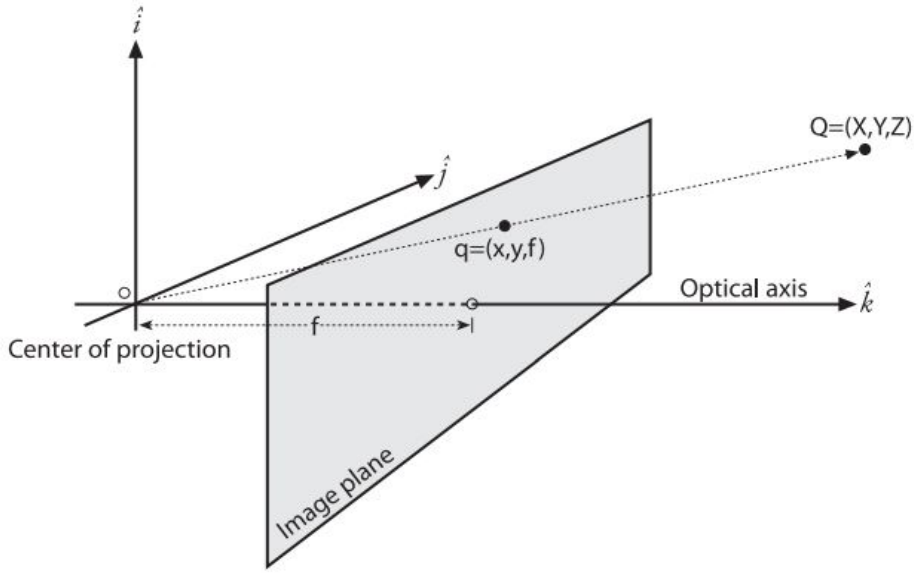
$$\frac{-x}{f} = \frac{X}{Z} \tag{2.1}$$

Where x is the objects image on the imaging plane, f is the focal length, X is the length of the object in the real world and Z is the distance from the camera to the object. Rearranging this shows that:

$$-x = f\frac{X}{Z} \tag{2.2}$$

The projected image is obviously upside down, which we know to be the case with the images perceived with our eyes as well. To make the math easier, it is useful to revert the position of the camera center of projection, and the imaging plane.

Figure 2.5 Reverted pinhole camera model. Image taken from [2].



This is shown in figure 2.5, and a similar triangular relationship forms:

$$x = f\frac{X}{Z} \tag{2.3}$$

The object projected to the imager is no longer upside down, as the negative sign disappears. The point at the intersection of the image plane and the Optical axis, k, is referred to as the *principle point*. From the figures it looks like the principle point is equivalent to the center of the image plane. However, for real cameras this is not the case, at least not exactly. Due to production errors the center of the imager will have some degree of deviation from the principle point in both x and y directions. These deviations are referred to as c_x and c_y. Thereby, the projection of a point P in the physical world, with coordinates (X, Y, Z) is projected onto the screen at some pixel location (x_{screen}, y_{screen}) given by the equations:

$$x_{screen} = f_x(\frac{X}{Z}) + c_x \quad y_{screen} = f_y(\frac{Y}{Z}) + c_y \tag{2.4}$$

The focal lengths f_x and f_y have pixel units, and is equal to the product of the physical focal length (unit is millimeters) and the size of the individual imager elements, s_x and s_y (units of pixels per millimeter). These four parameters define the camera and can be arranged into a 3x3 matrix called the camera intrinsic matrix. The projection of a point, P, in the physical world, with world coordinates (X, Y, Z), into the camera image plane on point p, is given by:

$$p = MP, \quad where \quad p = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad P = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{2.5}$$

Where M is the camera intrinsic matrix. Notice that the point p is given in homogeneous coordinates. The actual pixel coordinate can be recovered by dividing through p_3. This relation which maps the physical points to the points on the projection screen is called a *projective transform*.

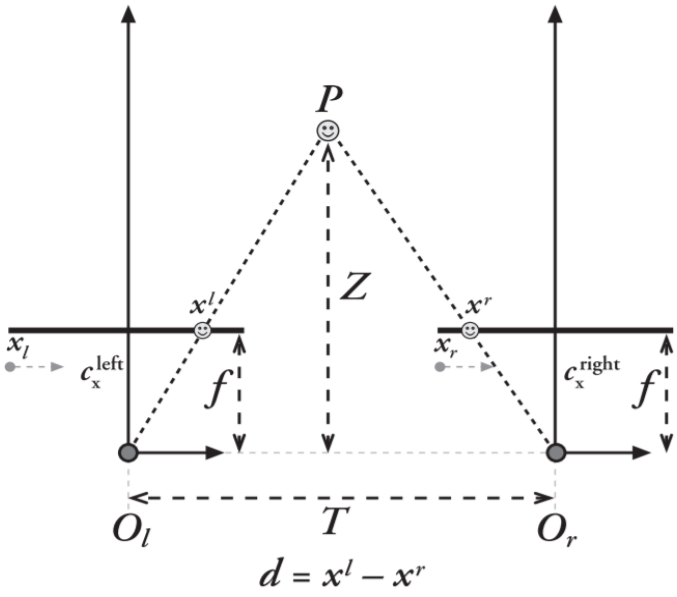
2.3 Stereo Imaging

The principle of Stereo Imaging in computer vision is the same as for us humans when we perceive the objects in close proximity. We have two eyes located at a fixed distance apart from each other. By observing the environment from different points of view at the same time, we are able to determine the 3D location of objects in the world. In other words, we are able to determine depth. Computers can emulate this by matching corresponding points in two images taken from different points of view. By these correspondences, knowledge about the cameras relative poses and the cameras intrinsic parameters, the 3D location of the points can be found by triangulation. In stereo imaging, the stereo image-pair is usually obtained through the use of a stereo camera, which has two lenses with separate image sensors.

2.3.1 Triangulation

Suppose we have an ideal stereo rig, that is, two cameras mounted at a known distance from each other with the exact same intrinsic parameters, the image planes exactly parallel, and every pixel row in one camera corresponding exactly with the pixel row of the other camera. Two equivalent triangles form in the model that enables the deduction of the 3D location of point P observed by both cameras.

Figure 2.6 Two ideal pinhole camera models, and the geometry forming the basis for depth measurement for an ideal stereo rig. Image taken from [2].



Let us denote the two triangles as $T_1(O_l, \text{P}, O_r)$ and $T_2(x_l, \text{P}, x_r)$. The height of T_1 divided by the baseline, T, is equivalent with the height of T_2 divided by $T - (x_l - x_r)$. This relation gives the equation for the depth measurement, Z.

$$\frac{B - (x^l - x^r)}{Z - f} = \frac{B}{Z} \mapsto Z = \frac{Bf}{x^l - x^r} \tag{2.6}$$

The depth is thereby inversely proportional to the disparity between these views, where the disparity is defined as $d = x^l - x^r$. This leads to a nonlinear relationship between these two terms. As a result the stereo vision system only has a high depth resolution for objects close to the camera.

This ideal scenario of having the image-pair row-aligned will in practice almost never be the case with real cameras. The aim is therefore to mathematically, rather than physically, align the image rows to look for correspondences. This process is called rectification, and in order to accomplish this it is necessary to understand the basic geometry of a stereo imaging system, called the Epipolar geometry.

2.3.2 Epipolar Geometry

Imagine two cameras observing the same point, P, in space. You want to find the projection of the point on both imagers, p_l and p_r, to calculate the disparity, and thereby the 3D location of P. You could use a feature descriptor to describe the point in image 1, and search every row and column (two dimensional search) in the second image to find the corresponding point. However, this would be computationally costly and very much

prone to error. The Epipolar geometry constrains this correspondence search from two dimensions (the whole second image) to one dimension (only a line in the second image).

Figure 2.7 Two pinhole camera models showing two conjugate epipolar lines and the location of epipoles. Image taken from [2].

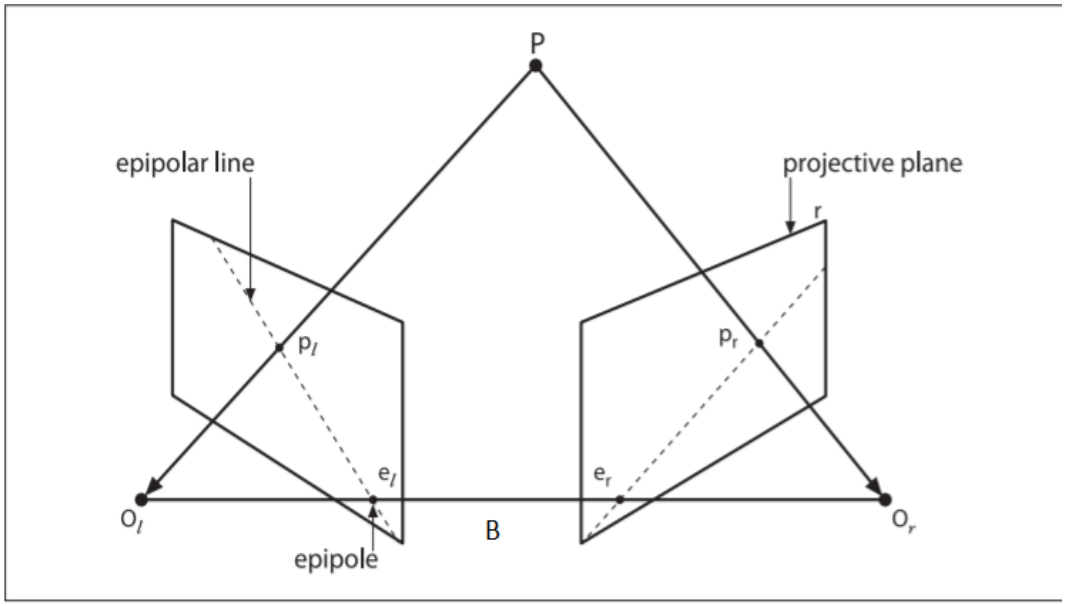


Figure 2.7 shows two pinhole cameras observing the same point, P, with camera centers O_l and O_r seperated by a vector called the Baseline, T. Point P observed by camera 1 forms on the pixel where line (P, O_l) intersects with the image plane. This projection of the point P on the left image plane is denoted p_l. From this single camera's observation of the point, there is one degree of freedom where that point could be in the world, anywhere along that line. Notice from the figure that the line (P, O_l) projected onto the second camera's image plane forms a line on the imager. This is called an epipolar line. The point in the left image, p_l, has a correspondence in the second image, p_r. The epipolar constraint says that p_r must lie on the epipolar line in the second image plane. This means that by knowing the epipolar geometry, finding a point in one image corresponding to a point in the other image, one only need to search along the epipolar line. All epipolar lines come in conjugate pairs, which means that for every epipolar line in the right image, there will be corresponding epipolar lines in the left image. The epipolar line just discussed in the right image, will have its conjugate in the left image. This epipolar line in the left image is given by the line that goes through p_l and an interesting point called the $epipole$. There is one epipole for each image, denoted e_l and e_r, and all epipolar lines pass through these points. The epipoles are located where the Baseline, the vector between the image centers, intersects with the image planes. One can think of the baseline as a hinge that has several planes passing through it. By twirling the hinge the planes will intersect the image planes at different locations. Wherever the plane intersects the image planes, there will be

conjugate epipolar lines.

2.3.3 The Essential and Fundamental Matrix

In order for computers to exploit these geometrical constraints, they have to be described mathematically. This is where the Essential Matrix, E, and the Fundamental Matrix, F, come into play. The Essential matrix contains information about how the cameras are related in space. That is, how one camera is rotated and translated with respect to the other camera in world coordinates. The Fundamental matrix combines this with information about the cameras intrinsic parameters, and thereby relates the cameras in pixel coordinates. The difference between these matrices is that the Essential matrix is purely geometrical, and relates the cameras in world coordinates. The Fundamental Matrix, on the other hand, relates points on the image plane of one camera in image coordinates (pixels) to the points on the image plane of the other camera in image coordinates.

Now, given a point P in the world seen by both cameras, the Essential matrix is the relation that connects the observed P_l and P_r of P by the left and right cameras. The aim is the compact equation:

$$P_r^T E P_l = 0 \tag{2.7}$$

Choosing the left camera as the reference frame, the observed point, P_r, is given by:

$$P_r = R(P_l - T) \tag{2.8}$$

Which can be rewritten as:

$$(P_l - T) = R^T P_r \tag{2.9}$$

The next step is the introduction of the epipolar plane. Using the fact that all points x on a plane with normal vector n passing through a point a obey the following constraint:

$$(x - a) \cdot n = 0 \tag{2.10}$$

and that the epipolar plane contains the vectors P_l and T, an equation for all points P_l through the point B and containing both vectors is:

$$(P_l - T)^T (T \times P_l) = 0 \tag{2.11}$$

Substituting equation "something something" into "somethig else" yields:

$$(R_T P_r)^T (T \times P_l) = 0 \tag{2.12}$$

The cross product of T and P_l can be expressed by defining the matrix S as the skew-symmetric matrix of T, Which leads to the final result:

$$P^T R S P_l = 0 \tag{2.13}$$

The product of RS is what defines the to be the essential matrix, E. By substituting for P_l and P_r:

$$P_l = \frac{p_l Z_l}{f_l} \quad and \quad P_r = \frac{p_r Z_r}{f_r} \tag{2.14}$$

and dividing everything by $\frac{Z_l Z_r}{f_l f_r}$, the relation between how the points are observed on the imagers is obtained.

$$p_r^T E p_l = 0 \tag{2.15}$$

To get the relationship between a pixel in one image, and an epipolar line in the other image, intrinsic information about the cameras must be introduced. The product of the camera intrinsic matrix, M, and a point, p, gives the point in pixel coordinates:

$$q = Mp \mapsto p = M^{-1} q \tag{2.16}$$

By substituting this, the equation for E expands to:

$$q_r^t (M_r^{-1})^T E M_l^{-1} q_l = 0 \tag{2.17}$$

By defining the Fundamental matrix as:

$$F = (M_r^{-1})^T E M_l^{-1} \tag{2.18}$$

The equation that relates the cameras in pixel coordinates becomes:

$$q_r^t F q_l = 0 \tag{2.19}$$

Using this equation to find epipolar lines can be easily done, by hand even, by fixing a point [x' y' 1] in image one, and then get the equation for a line in image two. The location of the epipoles can also easily be found by recalling that all epipolar lines intersect at the epipole. Let us denote the epipoler point as [x_e y_e 1], so that:

$$[x' y' 1] F [x_e y_e 1]^T = 0 \tag{2.20}$$

Because this equation always will be true no matter what x' and y' are, we see that:

$$F [x_e y_e 1]^T = 0 \tag{2.21}$$

The conclusion is that the epipoles can be found from the left and right nullspace of F. Following this, both F and E are 3x3 matrices but only have 7 degrees of freedom, and are of rank 2.

Figure 2.8 Three epipolar lines computed from the Fundamental matrix between a pair of images.

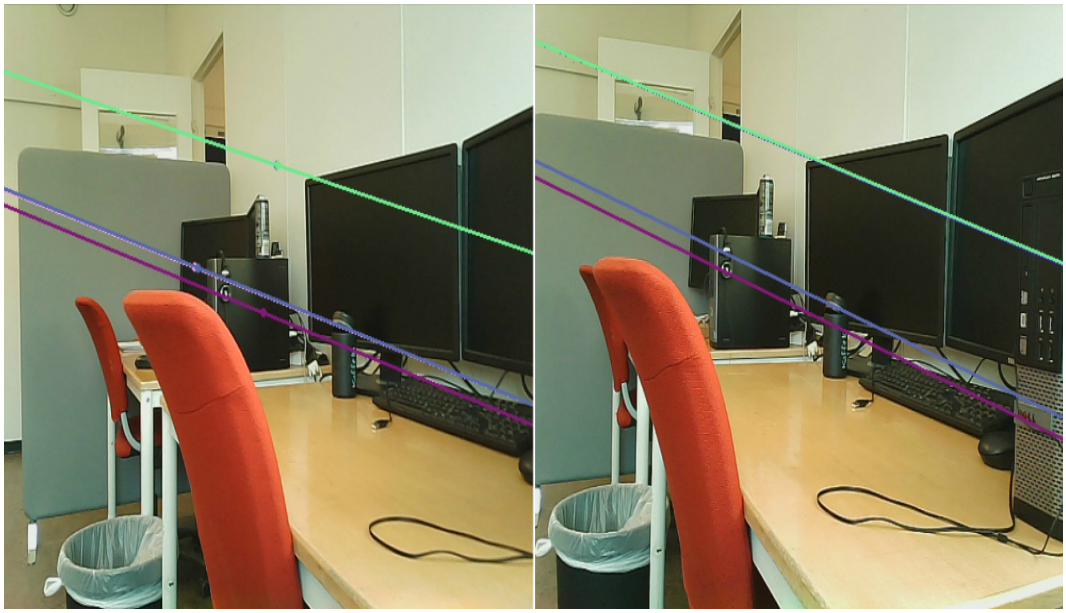


Figure 2.8 shows an image pair taken by the same camera at different times, of a static scene. The Fundamental Matrix was computed by finding the camera intrinsic parameters by calibration, and the Essential Matrix relating the different views in world coordinates. By choosing three points at random in the left image, the Fundamental Matrix gives the equation for the epipolar lines in the right image. The epipole in the left image is found as the eigenvector of the Fundamental Matrix with zero eigenvalue. Conjugate epipolar lines are plotted in the same colors.

2.3.4 Rectification

Computing the stereo disparity between image pairs is easiest when the images planes align exactly, as seen in the Triangulation section. However, this is almost never the case, as stereo rigs will almost never have perfectly co-planar, row-aligned image planes. The process of rectification is to mathematically make the image planes perfectly co-planar and row-aligned. Rectification will not only make the search for corresponding pixels more efficient, but it will also reduce the chances of getting wrong matches.

Traditional Rectification: Hartley's Algorithm

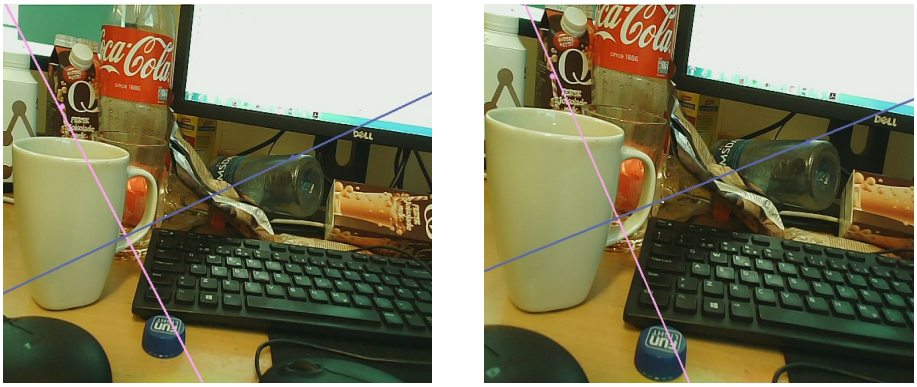
Hartley's algorithm [2] rectifies the image-pair by finding a homography that maps the epipoles to infinity. This causes the epipolar lines to intercept at infinity, making the epipolar lines parallel, and consequently making corresponding image points being located on the same image row. This is accomplished by simply matching points between two im-

age pairs. Because the camera intrinsics information is implicitly contained in the point matches, the only input needed is the Fundamental Matrix. As seen in the previous section, the Fundamental Matrix can be computed by only having to find good point matches between the image pair. This makes Hartley's algorithm able to perform stereo rectification online by only having to find point matches between the images. However, Hartley's algorithm, and other traditional rectification schemes, fail when the epipoles are located on the image planes. Mapping the epipoles to infinity when they are located on the image plane will result in infinitely large images. Even when the epipoles are not on the image planes, but very close, the rectified images will be absurdly large.

Generalized Rectification

"A simple and efficient rectification method for general motion" [3] deals with all camera motions, meaning it handles rectification of an image-pair regardless of where the epipoles are located. The novelty of their approach is reparameterizing the image with polar coordinates around the epipoles. Also, the epipolar lines are divided into *half* epipolar lines. Consider the image-pair in figure 2.9.

Figure 2.9 Image-pair where the epipoles are located on the image planes

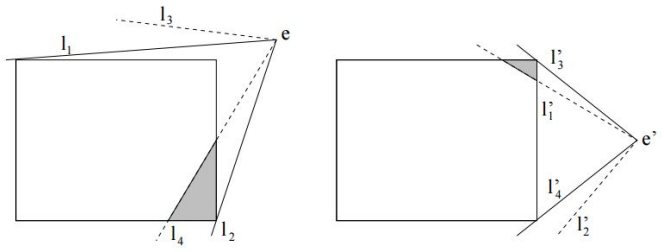


(a) Image 1 **(b)** Image 2

The epipoles are located where the epipolar lines intersect, on the image plane. The search for correspondences between the images should be along the same half of each conjugate epipolar line, where the epipole is the dividing factor.
The algorithm consists of the following steps:

Determining the common region The first step is to determine the common epipolar lines. This is done by first finding the extremal epipolar lines for both images. These are defined as the epipolar lines that touch the outer image corners. If the epipole is located on the image planes, any arbitrary line can be chosen as the starting point. Once the extremal epipolar lines have been found for both images, the common region is determined as shown in figure 2.10.

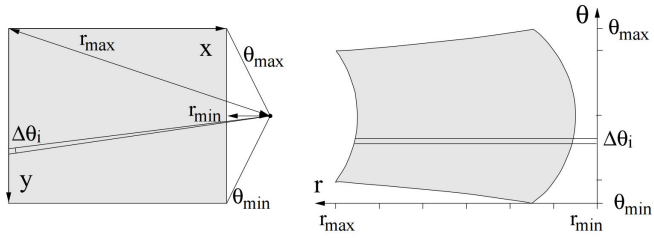
Figure 2.10 Determination of the common region. The extreme epipolar lines are used to determine the maximum angle. [3]



Determining the Distance Between Epipolar Lines The second step is to determine the minimum distance between two consecutive epipolar lines. This is done to avoid losing pixel information when doing the rectification. The worst case pixel is always located on the image border opposite to the epipole, so that the distance between two consecutive epipolar lines at this location should always be at least one pixel. This is done for both images.

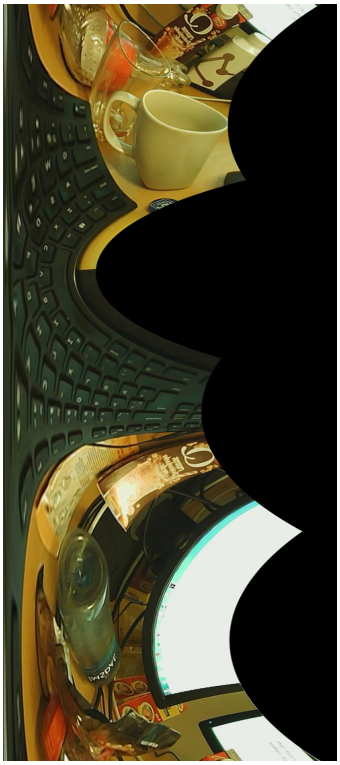
Constructing the Rectified Image The third, and final step, is constructing the rectified images. They are built up row by row, transforming the image coordinates from Cartesian(x, y) to angular (θ, r). The maximum and minimum angles are given by the angle from the epipole to the nearest image corners, while the maximum and minimum r-values are given by the distance from the epipole to the furthest and closest image border. This is made more clear by figure 2.11.

Figure 2.11 Transformation of the image from Cartesian to polar coordinates. The θ axis is nonuniform so that every epipolar line has an optimal width. [3]

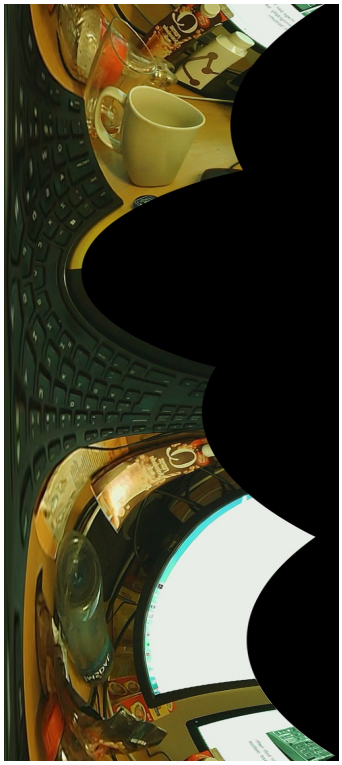


This rectification algorithm promises that the size of the rectified images are the minimum of what can be achieved without pixel compression. Figure 2.12 shows a rectified image pair computed by this algorithm, where the epipoles are located on the image planes.

Figure 2.12 Rectified image-pair where the epipoles are located on the image planes



(a) Rectified Image 1

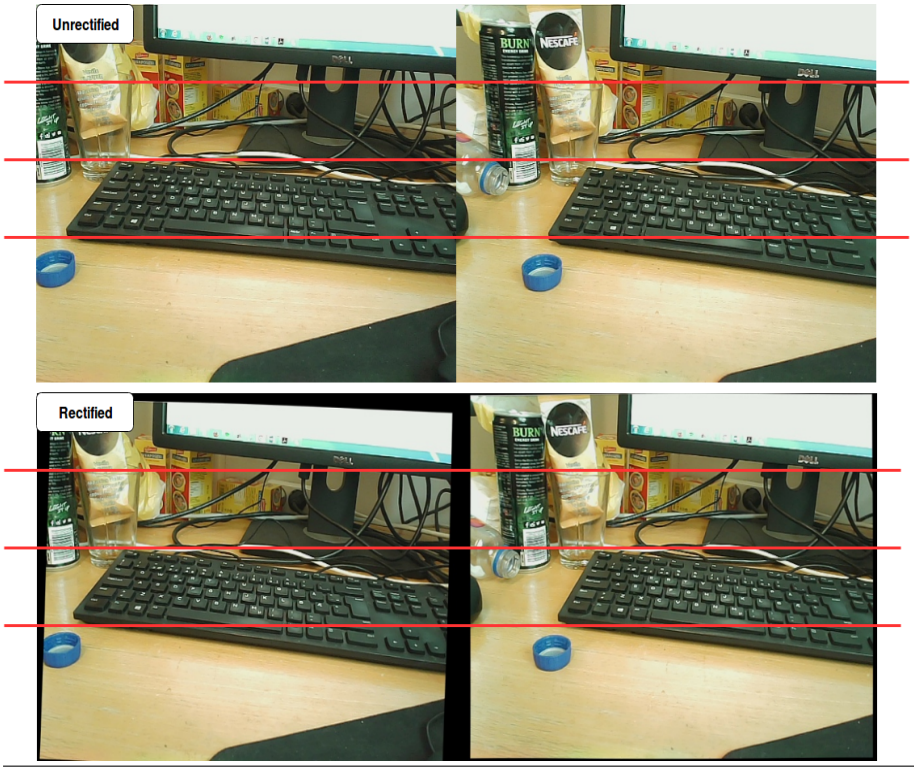


(b) Rectified Image 2

2.3.5 Stereo Matching

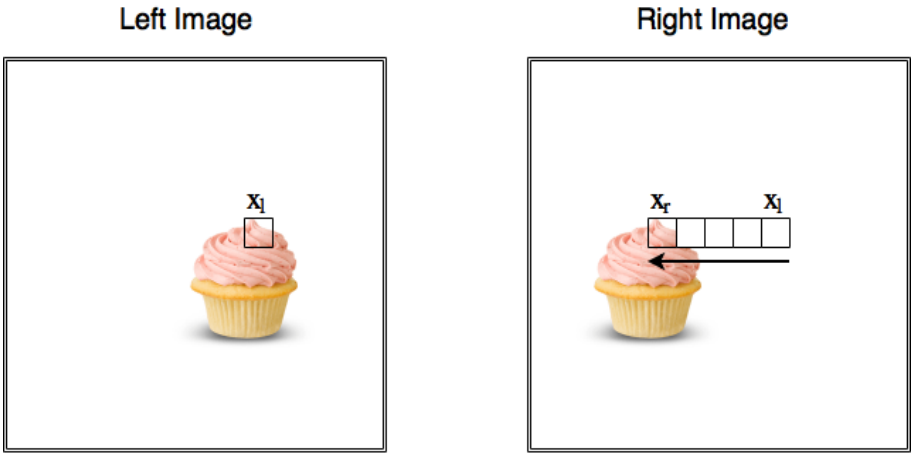
Stereo matching, also known as stereo correspondence, is the process of matching the corresponding 3D point in the two different camera views. Most stereo matching algorithms assumes that the input stereo image-pair is rectified, so that corresponding points can be found on the same lines, as in figure 2.13.

Figure 2.13 Distorted image-pair, and the image-pair after undistortion and rectification.



To illustrate correspondence search with an easy example, consider figure 2.14 below.

Figure 2.14 Correspondence search from left to right camera view



The depth measurement of the cucpake-patch in the left image coordinates can be triangulated from the disparity measurement, $d = x^l - x^r$, or $d = x^l - x^r - (c_x^{left} - c_y^{right}$ if the camera intrinsics are known. If the camera intrinsics are unknown, depth can only be measured up to a scale factor. As seen in figure 2.14, the current part of the left image that is being considered is the top right corner of the cupcake, located at image coordinates (x_0, y_0). Because the second image is the right view of the cupcake, the corresponding cupcake patch must lie to the left of right image coordinate (x'_0, y'_0). Thus, the search for the corresponding image patch is conducted to the left of this point. Which of the right image patches that matches the current left image patch can be decided by calculating the cost for each patch-pair. The cost might be the sum of absolute differences for every pixel-pair. The right patch with the minimal cost can then be regarded as the match. The cupcake example is used for the sake of it being a very simple example, however, the resulting disparity map would be of very low resolution, due to using large patches instead of individual pixels. For dense disparity map computation, every individual pixel in the left image should be matched with every individual pixel in the right image. The cost function could then be the difference in intensity of the current pixels being looked at, a sliding window around the current pixel being looked at, where the cost could be the sum of absolute differences of all pixels in the window, or the similarity between the current pixels descriptors.

There are many challenges when doing stereo correspondence. In low-textured scenes, such as in an indoor hallway, finding the correct matches can be very challenging. This is due to the fact that many pixels in the same regions might have very similar intensities, and the point which should be a correct match might look dissimilar due to illumination changes.
Disparity measurements are obviously only possible where the two images overlap, and the regions of the images that don't overlap can be a source of bad matches.
Another challenge is *occlusions.* Occlusions are parts of the scene that are not visible from one of the camera views perspective. Take these images, figure 2.15, from the Middlebury dataset [4]:

Figure 2.15 Rectified stereo-image pair from the Middelbury dataset.

(a) Left image

(b) right image

The occluded parts of this stereo pair can be seen in figure 2.16.

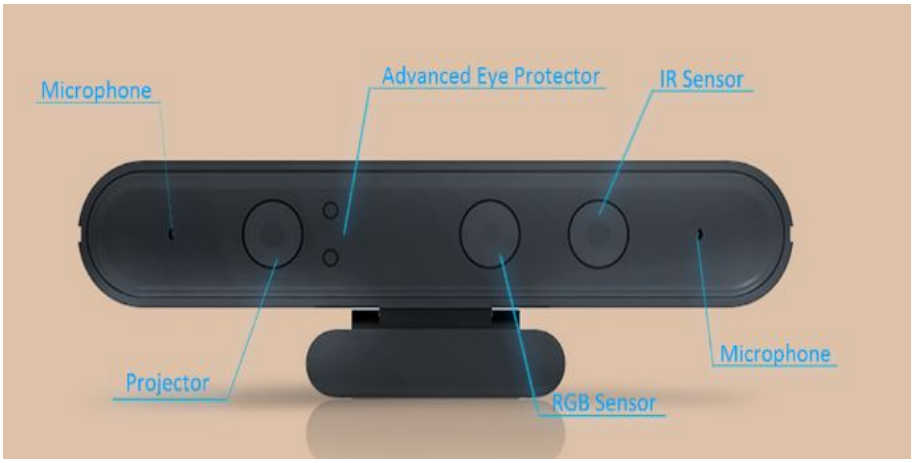
Figure 2.16 Occluded regions of teddy image pair [4].

To deal with noise, occlusions and low-textured areas, post-filtering is used to deal with bad matches. Some post-filtering techniques are, but not limited to, crossmatching, uniqness ratio and texture thresholding. Crossmatching is simply to do the matching both ways, from left to right and right to left, the latter being the crossmatch. If the crossmatch is equal, or close, to the considered left image pixel, the match is considered good. If not, it is considered a bad match.

2.4 Depth Sensors

Depth sensors, also known as range cameras, are devices made for producing a 2D image with pixel values corresponding to the distance from the sensor. There are many different techniques to accomplish this. Some of them, but not limited to, are stereo vision, time-of-flight and structured light[18]. Stereo vision was just explored in the previous sections and is a very low cost solution. However, the quality of the depth maps from stereo vision depends highly on a number of factors, one of them being the matching algorithm. For stereo vision, better depth maps mean longer computational time, and for some applications this trade off is not acceptable. Time-of-flight sensors are, on the other hand, extremely fast, and well suited for real-time applications. They operate by sending out light pulses that get reflected by objects in the scene. The depth is then determined by the camera lens gathering the reflected light. The speed of light is known, and the depth is found from the time it took the light to be reflected back to the sensor. Time-of-flight sensors might experience problems when a lot of background light is present. The sensor used in this project is an Orbbec Astra Pro[19]. It uses the software technology of Kinect[10], Microsofts depth sensor made for the video games console, Xbox 360, in 2011.

Figure 2.17 Orbbec Astra Pro



The Orbbec Astra Pro techical specifications are:

- **Size/Dimensions:** 160 x 30 x 40 mm
- **Weight:** 0.3 kg
- **Range:** 0.6-8m
- **Depth Image Size:** 640x480 (VGA) 16bit @ 30FPS
- **RGB Image Size:** 1280x720 @ 30FPS (UVC Support)
- **Field of View:** 60°horiz x 49.5°vert. (73°diagonal)
- **Data Interface and Power:** USB 2.0
- **Microphones:** 2
- **Operating Systems:** Windows, Linux, Android
- **Software:** Orbbec Astra SDK + OpenNI

The Kinect sensor uses a concept from structured light[18], called light coding. This is the process of projecting a known pattern of dots onto a scene. The pattern is projected by an infrared projector on the Kinect, and read through an infrared camera positioned at a known distance from the projector. Because the pattern projected is known by the sensor, it can search for correlations between the internal image of the pattern and the one reflected by the scene. By matching dot clusters it is able to determine the 3D position of the projected dots. This is accomplished by triangulation. The output will be a depth map[16]. A depth map is an image where the intensities of each pixel is determined by how far it is from the sensor in 3D space. Different colors/intensities represent different depths, or distances from the sensor.

Figure 2.18 Depth map taken with Orbbec Astra Pro

As seen from figure 2.18, the lighter the intensity of the pixel, the further the point is from the camera, and vice versa. Another interesting thing to note from figure 2.18 is that the infrared light hitting the computer screens in the background is not reflected properly, thus giving the pixels a darker intensity.

2.5 Visual Simultaneous Localization and Mapping

Simultaneous Localization and Mapping[11], or SLAM for short, is the process of constructing a map of an unknown environment while at the same time knowing where an agent is within it. In SLAM it is important to strive towards having a global, consistent estimate of the robot path. This involves keeping track of a map of the environment, even when the map is not actually needed for the application. The reason for this is that a robot needs to know if it revisits a previously explored area. When the robot tracks its location relative to a set of $landmarks$, there may be errors affiliated with these landmarks. The accumulation of these errors over a long trajectory will result in a large error. This error accumulation is called $drift$. When a robot then revisits and recognizes a previously explored area it can use this information to correct the path trajectory and map error. This is called *loop closure*. There are several techniques and types of sensors used for solving the SLAM problem. In recent years there has been a lot of research about using visual sensors, like a depth sensor, to solve the SLAM problem. This is called Visual Localization and Mapping, or VSLAM for short.

2.6 KinectFusion

KinectFustion[10] is an algorithm developed by Newcombe et al. in 2011 that enables a user to create detailed 3D reconstructions of an indoor scene by using a Kinect sensor in real time. The depth data from the Kinect sensor is used to track the 3D pose of the sensor and create a model of the unknown environment simultaneously.

The system consists of four main stages, which will be elaborated momentarily. The four main stages are:
a) Depth Map Conversion
b) Camera Tracking
c) Volumetric Integration
d) Raycasting
In order for the system to operate in real time, the algorithms that makes up KinectFusion have been designed from the ground-up for parallel execution on the GPU using the CUDA language[20]. CUDA, invented by NVIDIA, is a parallel computing platform and programming model that enables software engineers to use C, C++ and Fortran to use GPU resources.

2.6.1 Depth Map Conversion

The first step of the KinectFusion algorithm is to extract the current depth map from the sensor. The raw depth data is then filtered with a bilateral filter to remove noise. The bilateral filter is able to remove noise while preserving edges in the image by replacing the intensity of each pixel by a weighted average of its surrounding pixels. The filtered depth map is then converted into actual 3D coordinates with metric units. By creating a vertex for each pixel in the depth map, where x is the x position of the pixel, y is the y position of the pixel, and z is the depth value, and converting these values to metric units, the resulting product is a vertex map, also known as a point cloud. By calculating the cross product of a vertex and its surrounding pixels, the normal vector for each vertex is found. The result is a model consisting of two maps; a vertex map and a normal map. Here each CUDA thread operates on a separate pixel in the depth map, computing the vertex map in parallel.

2.6.2 Camera Tracking

Suppose that the algorithm is not in its first iteration. Then the current model has been constructed from previous iterations of the algorithm. There are now two pairs of the vertex and normal maps. One pair from the current model, and one pair from the new depth map. The next step of KinectFusion is running the Iterative Closest Point (ICP) algorithm on these maps. The ICP algorithm is popularly used to align 3D shapes. In KinectFusion it is used to track the camera pose of each new depth frame. By rotating and translating the new depth map with the current model, the point clouds will be aligned. This 6 degree of freedom transform that makes the point clouds aligned is what we are interested in. This rotation and translation that minimizes the error between the two point clouds, gives the cameras position relative to the current model. The important first step of ICP is to choose the best starting position between the two models to find point correspondences. KinectFusion chooses the pose of the previous depth map as the starting position for ICP,

because we assume that the sensor started there and has moved a small distance. A typical ICP algorithm uses point-to-point error to find corresponding point-pairs. This means that it will look at a point in one of the pointclouds, and find the point on the other point cloud that is closest, and make them a point-pair. After doing this for every point, it will find a translation and rotation that aligns all the point-pairs, save this Transformation and repeat. However, because the KinectFusion runs in real time, it assumes that the changes between frames are small. It gets corresponding point pairs between the model and the previous depth map by using projective data association. The way projective data association works is by looking at both pointclouds from the same point of view, and then "shooting bullets" through the overlaying point clouds. If the "bullet" passes through a point in the first cloud and then a point in the second cloud, these are corresponding point-pairs. Then ICP is run on these corresponding point-pairs, estimate the error, find the transformation that minimizes this error, and then repeat ICP until a good match is found or until it has used too many attempts.

2.6.3 Volumetric integration

By using ICP to find the pose of the camera at new depth measurements, a global camera pose, T_i, can be updated with an incremental transform calculated per iteration of ICP. This makes every depth measurement able to be converted from image coordinates into a single global coordinate space. From the previous step, we know how the current model and the new depth data fit together. KinectFusion merges these together in a fixed size 3D volume. The volume in subdevided uniformly into a 3D grid of voxels, where vertices in global coordinates are integrated into the voxels using Trancuated Signed Distance Functions(TSDF). The value each voxel inhibits is given by the TSDF, and is specifying the distance to a surface. The value will be positive in front of the surface and negative behind the surface. The zero-crossings define where the surface is actually located. Suppose that the algorithm is not in its first iteration. Then the current model will already have a volumetric TSDF representation. By making a volumetric TSDF of the new depth measurments the two volumetric TSDFs can be merged into a single volumetric TSDF using weighted averaging. The result is the new current surface representation. The full 3D voxel grid is allocated on the GPU as aligned memory. This is not memory efficient, as it takes up 512MB of memory, but it is very speed efficient.

2.6.4 Raycasting for Rendering and Tracking

A GPU-based raycaster is implemented to generate views of the surface within the volume. In parallell, each GPU thread walks along a single ray and extracts the position of the surface by looking for zero-crossings in the TSDF. This process extracts the points that makes up the pointcloud of the environment. By raycasting from the global pose of the camera, a more globally consistent and less noisy environment representation is achieved. When new depth maps arrive from the live depth sensor, the globally consistent raycasted view of the model can be used as reference frame for the next iteration of ICP. By aligning the new depth map with this volume, tracking can be achieved without using live depth maps frame-to-frame.
Every step of the algorithm happens in real-time at about 30 times per second.

2.7 Kintinuous

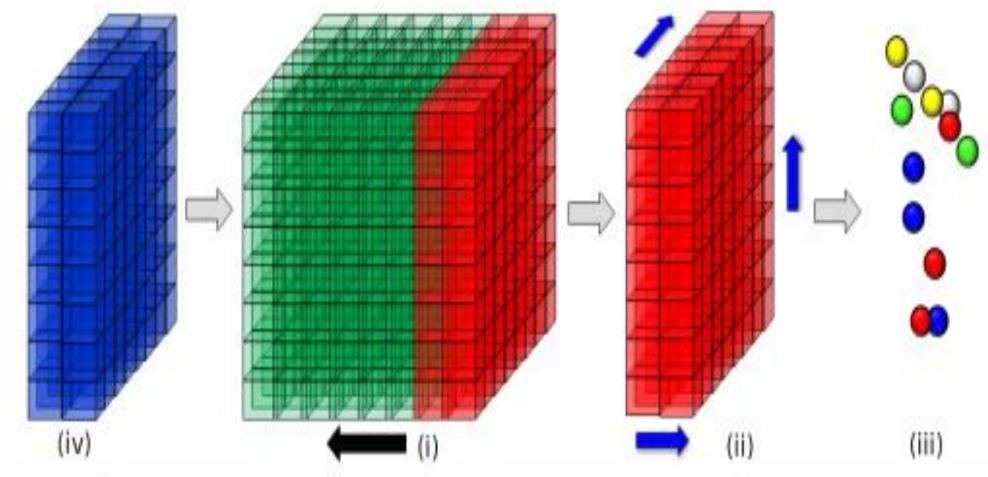
Kintinuous[5] is an extension to the KinectFustion algorithms that permits dense mesh-based mapping of extended scale environments in real time.

While the KinectFusion algorithm restricts the tracking and surface reconstruction to the region around the point of initialisation of the TSDF, Kintinuous permits the area mapped by the TSDF to move over time. The process is as follows:

1. Check how far the camera is from the origin
2. If above a specified threshold, b, virtually translate the TSDF to center the camera
   (a) Extract surface from region no longer in the TSDF and add to pose-graph.
   (b) Initialise new region entering the TSDF as unmapped.

The threshold, b, is the distance in meters in all directions from the current origin that the camera may move before the TSDF recenteres. If the camera never travels beyond this threshold, and while the camera remains within this region specified by b, Kintinuous functions exactly like KinectFusion. However, since Kintinuous permits this TSDF volume to travel , while keeping track of the cameras position within the scene, Kintinuous is an RGB-D based SLAM system.

Figure 2.19 Figure illustrating the four main steps of the Kintinuous algorithm. Taken from [5]. (i): Camera motion exceeds the movement threshold, b. The voxels in red now lies outside the boundary of the TSDF. (ii) The red volume slice is then raycast for point extraction. (iii) The Point cloud is extracted and fed into Kintinuous mesh triangulation algorithm. (iv) A new region enters the volume of the TSDF.



Kintinuous way of dealing with drift is by implementing visual loop closure using a $bag - of - words$ (BoW) model[21]. Image features are treated as words, and thereby

creating a visual vocabulary. When detecting a loop closure, a pose constraint is calculated between the matching RGB images and propagated into the TSDF virtual frame to adjust the poses.

CHAPTER 3

Design and Implementation

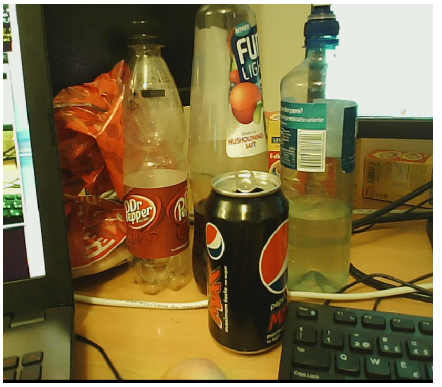
The Kinect sensors ability to produce high quality depth maps at a frequency of 30Hz is a key component of making the Kintinuous algorithm a robust Visual SLAM system. However, the algorithm is limited by the effective range of the Kinect sensor. The minimum range of 0.6 meters makes the sensor unable to produce high resolution depth maps for objects located within this range. This section gives an overview of the multiview geometrical reconstruction algorithm produced to compensate for the limited range of the Kinect sensor. Section 3.1 will provide a general overview of the algorithm, as well as the interaction between the systems main components. Sections 3.2 - 3.5 give a more detailed description of each component. For reasons explained later in the thesis, the depth map fusing was never achieved. Section 3.6 provides what was planned for the fusing of the depth maps.

3.1 System Overview

The algorithm developed applies what is applicable from Stereo Vision theory, and uses what is already at disposal; the single RGB-camera found on the Kinect, and the tracking data provided by Kintinuous. To compute a depth map from normal RGB-cameras, two different views of the scene is required, as well as the baseline separating the camera at the capturing of the two images. Kintinuous makes this possible, as it provides a dense pose graph containing the cameras current translation and rotation with respect to the initial camera position. Furthermore, having access to the computed orientation of the camera at all times, consequently provides the Essential Matrix relating the image taken at time t_n to all images taken previously, when keeping track of which orientation data belongs to which previous image. Computing the Fundamental Matrix relating two images taken at different times in pixel coordinates is then only one step away, namely calibrating the Kinect RGB-camera to obtain its intrinsic parameters. Having the Fundamental Matrix

Figure 3.1 The depth map produced by the Kinect sensor when held at approximately 0.5 meters from distant objects, and the corresponding RGB image.

(a) Depth map



(b) RGB image

at disposal makes rectification of the images possible, thus making pixel correspondences searches much faster and more reliable. This is the essence of the approach conducted by the mutliview projective reconstruction algorithm, which can be outlined by the following steps:

1. Save two images as Keyframes, separated by the desired baseline, along with the orientation of the camera at the capture of each Keyframe.

2. Find the Fundamental Matrix relating the two Keyframes in pixel coordinates.

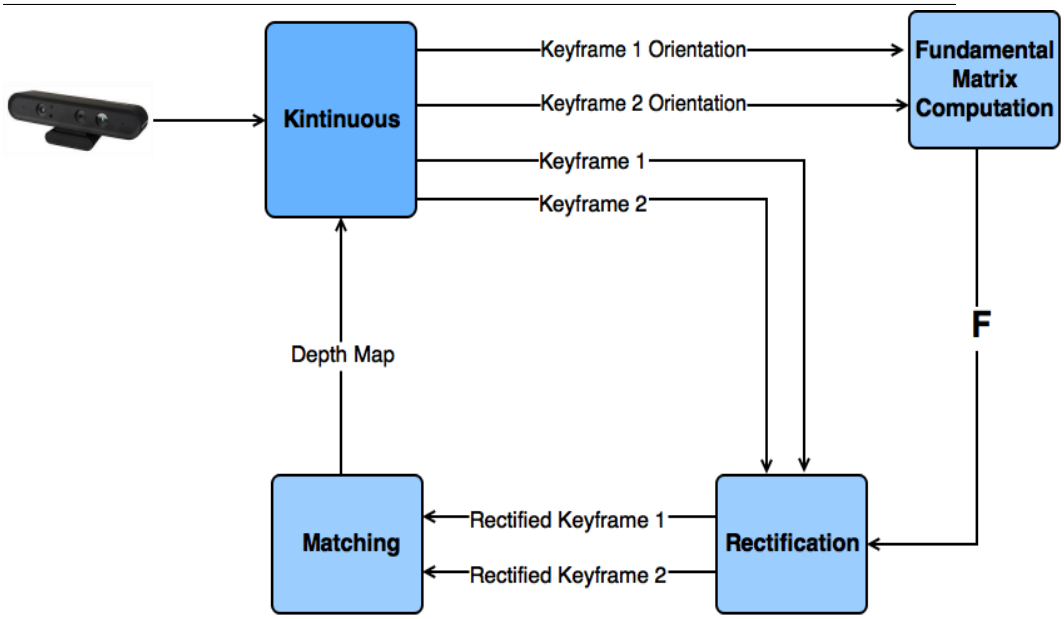
3. Rectify the Keyframes to make the epipolar lines row-aligned.

4. Produce a dense disparity map by matching corresponding pixels.

5. Fuse the depth map with Kintinuous.

”Keyframe” is the denotation given to images chosen for the computatation of the depth map. The system as a whole, and its individual key components, can be seen in figure 3.2.

Figure 3.2 System overview



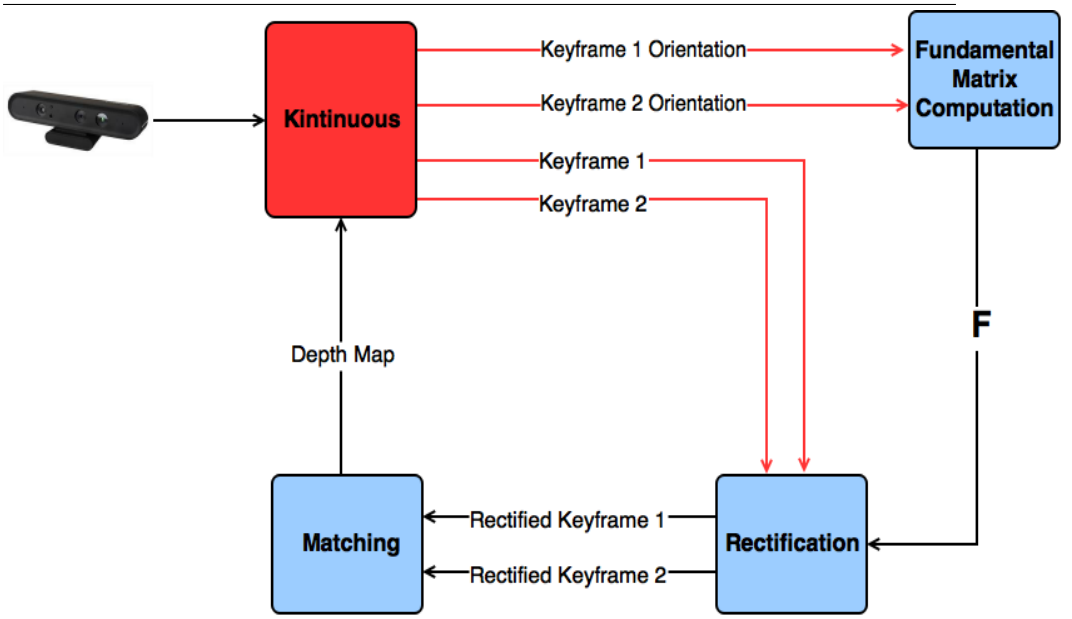
The system is implemented as a new class in Kintinuous, called "*StereoDepth*", and gets spawned as one of Kintinuous threaded system components when Kintinuous initializes. Every system component thread in Kintinuous inherits from the same superclass, called *ThreadObject*, which provides a series of virtual methods. One of these virtual methods is the *process()* method, which gets called in an infinite loop, until it either returns an error or the system is shut down.

3.2 Keyframe Acquisition

The Keyframe acquisition component of the system is highlighted in figure 3.3, and is responsible for the following:

- Saving the Keyframes along with the camera pose at the capture of said Keyframes
- Ensuring that the camera position of each Keyframe is separated by the desired baseline.
- Finding the direction of the camera from the first Keyframe to the other.

Figure 3.3 System overview, highlighting the Keyframe acquisition component



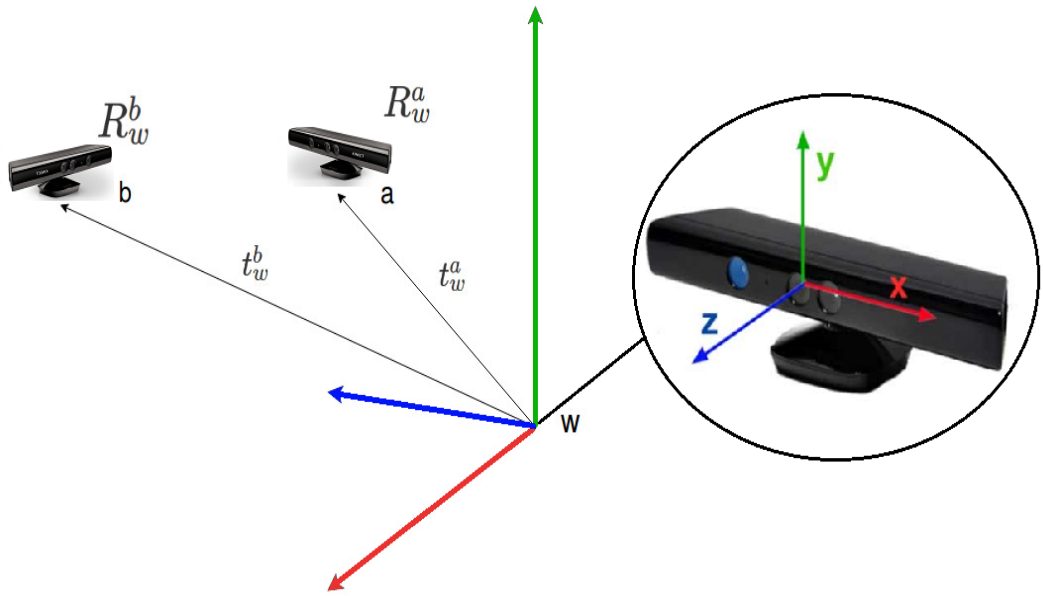
3.2.1 The Keyframes and Their Orientation

Whether an image is chosen as a Keyframe is decided by the following conditions:

- If there are **no** Keyframes saved for processing:
  - save the current RGB image as a Keyframe, along with the current camera orientation.
- If there is **one** Keyframe saved for processing:
  - When the camera has moved from Keyframe 1's orientation to form the desired baseline, save the current RGB image as a Keyframe along with the current camera orientation.

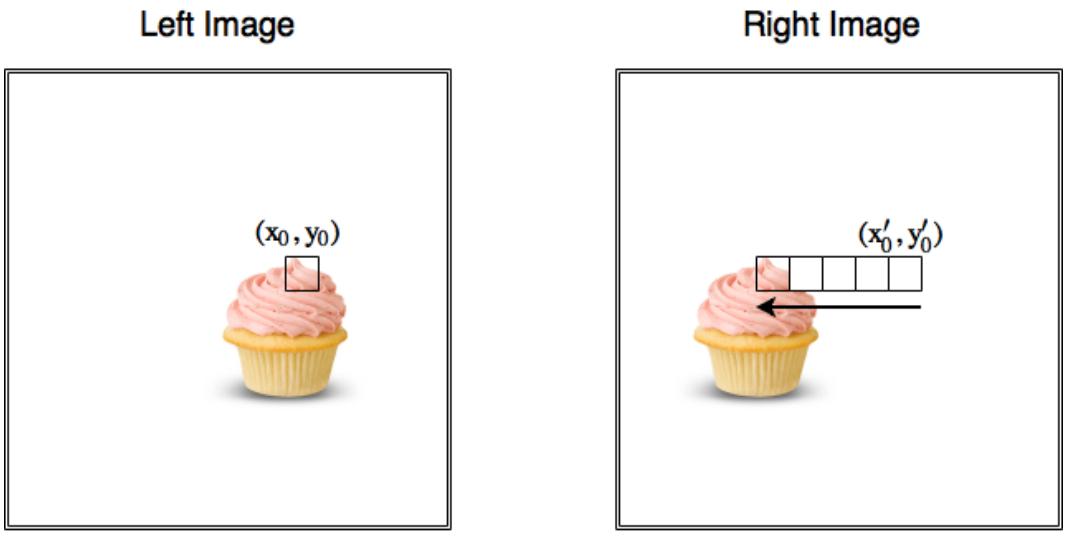
The orientation of the camera when a Keyframe is saved is in the form of a translation vector and a rotation matrix. These are pulled directly from a dense graph created by Kintinuous, consisting of the camera poses. The dense pose graph is updated by Kintinuous after each iteration of the ICP algorithm.
When Kintinuous initializes, the current orientation of the camera is then defined as the world reference frame. Every rotation matrix and translation vector pulled from the dense pose graph is then the rotation and translation of the camera with respect to the world reference frame. This is illustrated in figure 3.4.

Figure 3.4 The world reference frame in Kintinuous, defined by the coordinate system of the Kinect.



When there are no Keyframes saved for processing, the current RGB image from the Kinect is stored in an OpenCV image container, along with the last entry of the dense pose graph, becoming Keyframe 1. Following this, every new entry of the dense pose graph is compared to the translation vector of Keyframe 1. If the magnitude of the vector between the translation vector of Keyframe 1 and the current translation vector of the dense pose graph is larger than, or equal to the desired baseline, the current RGB image will be stored as Keyframe 2, along with the current entry of the dense pose graph.
The final purpose of this system component is to compute which most prominent direction the Kinect took between the capture of each Keyframe. The reason for this is to make the correspondence search faster and less prone to error. To illustrate this, consider the rectified images of the delicious vanilla cupcake with strawberry frosting in figure 3.5.

Figure 3.5 Reduced correspondence search when knowing which direction the Kinect took between the capture of the Keyframes



By knowing that one of the images is the leftside view, and the other the rightside view, the correspondence search only needs to be conducted in one direction. Considering pixel (x_0, y_0) in the left image, the corresponding pixel in the right image must be to the left of pixel (x'_0, y'_0). Consequently, if Keyframe 1 is found to be the leftside view, and Keyframe 2 the rightside view, the correspondence search is constrained to be conducted to the left. Similarly, if Keyframe 1 is found to be the rightside view and Keyframe 2 the leftside view, the correspondence search is constrained to be conducted to the right. This is also the case for forward, backwards, upwards and downward motion of the camera.
Computing which most prominent direction the camera went, can easily be done by looking at the x, y and z values of $t_{Keyframe2} - t_{Keyframe1}$. The largest absolute value of x, y and z shows which axis the camera travelled the furthest along, and the sign of the axis-value gives the direction.
Thus, the Keyframe Acquisition system component takes the following input:

- RGB-images from the Kinect.
- Camera poses from the dense pose graph created by Kintinuous.

and outputs the following:

- Keyframe 1 and Keyframe 2, and their orientations with respect to the Kintinuous world reference frame.
- The most prominent direction the camera took between the capture of each Keyframe.

3.2.2 The Desired Baseline

The baseline between two camera views is the translation between each camera's center, and is directly affecting the depth measurement when triangulating, as seen in section 2.7.1. The choice of baseline in this system is directed towards matching the depth accuracy of Kinect at its lowest range, which is 1.5mm. The depth accuracy for a stereo system can be found by the following formula [2]:

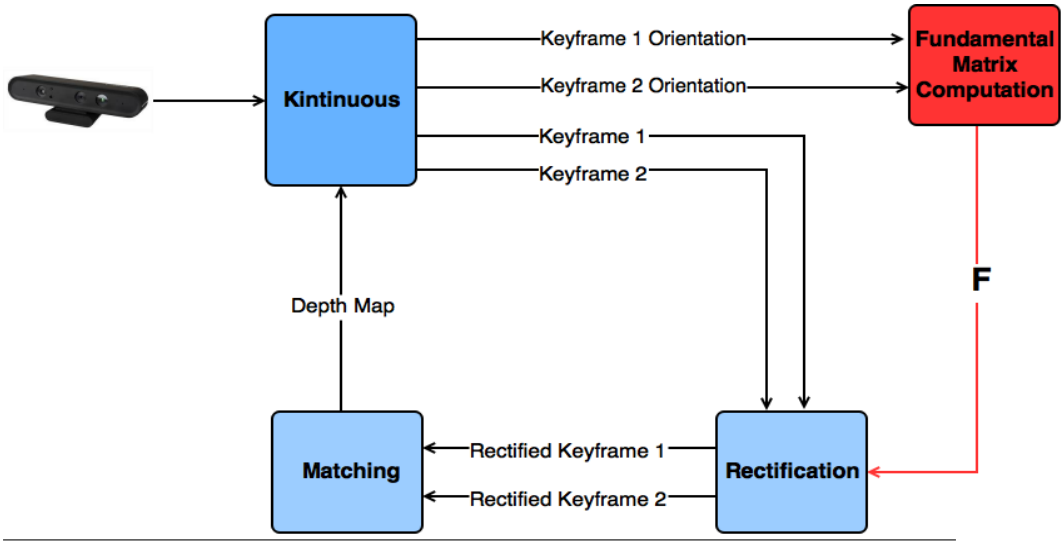
$$\Delta z = \frac{z^2}{fb}\Delta d \tag{3.1}$$

where Δz is the depth accuracy for which depth, z, f is the focal length and Δd is the disparity error. For the specific depth sensor used in this project, with a focal length of 11 mm, the desired depth accuracy for objects located closer than 0.6 meters is obtained with a baseline of 5 cm.

3.3 Fundamental Matrix Computation

The Fundamental Matrix Computation system component is highlighted in figure 3.6. Its purpose is to compute the Fundamental Matrix relating the Keyframes in pixel coordinates, which is needed for the rectification.

Figure 3.6 System overview, highlighting the Fundamental Matrix Computation component

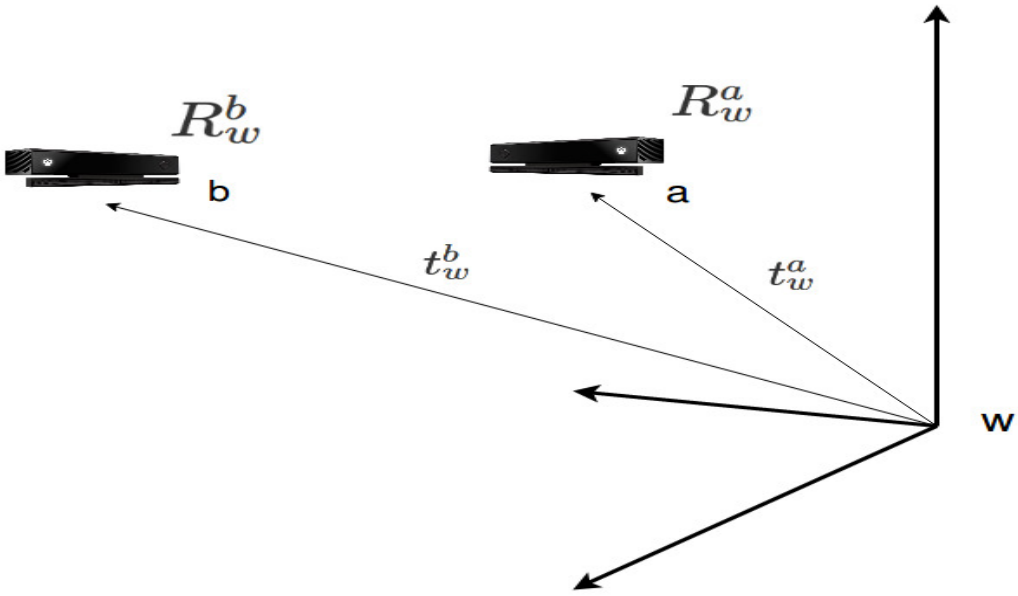


The Fundamental Matrix is given by

$$F = (M_r^{-1})^T E M_l^{-1} \tag{3.2}$$

where $M_r = M_l = M$, since both Keyframes are taken by the same camera, with the same intrinsic parameters. The Essential Matrix relating the two Keyframes in geometrical coordinates is computed by using the orientation matrices outputted by the Keyframe Acquisition component. These orientation matrices are the poses of the camera at each Keyframe with respect to the Kintinuous world reference frame, as illustrated by figure 3.7. The rotations and translations of Keyframe 1 and Keyframe 2 are denoted $[R_w^a t_w^a]$ and $[R_w^b t_w^b]$, respectively.

Figure 3.7 The orientation of the camera at the acquisition of Keyframes a and b



The orientation of Keyframe 1 is chosen as the new reference frame, and the Essential matrix is obtained. The Essential Matrix relating the Keyframes in geometrical coordinates is illustrated in figure 3.8.
Having the two rotation matrices, R_w^a and R_w^b, R_a^b is found as:

$$R_a^b = R_w^b * (R_w^a)^T \tag{3.3}$$

The translation between the Keyframes is found as:

$$t_a^b = t_w^b - t_w^a \tag{3.4}$$

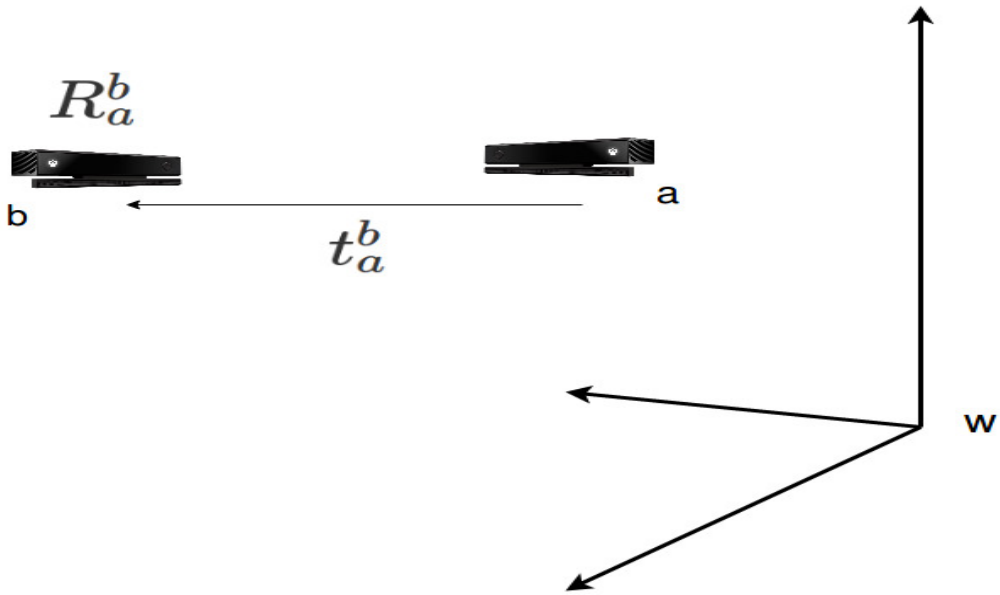
The Essential Matrix relating the Keyframes in geometrical coordinates is then found:

$$E = R_a^b \times t_a^b \tag{3.5}$$

Lastly, the Fundamental Matrix is then obtained as:

$$F = (M^{-1})^T E M^{-1} \tag{3.6}$$

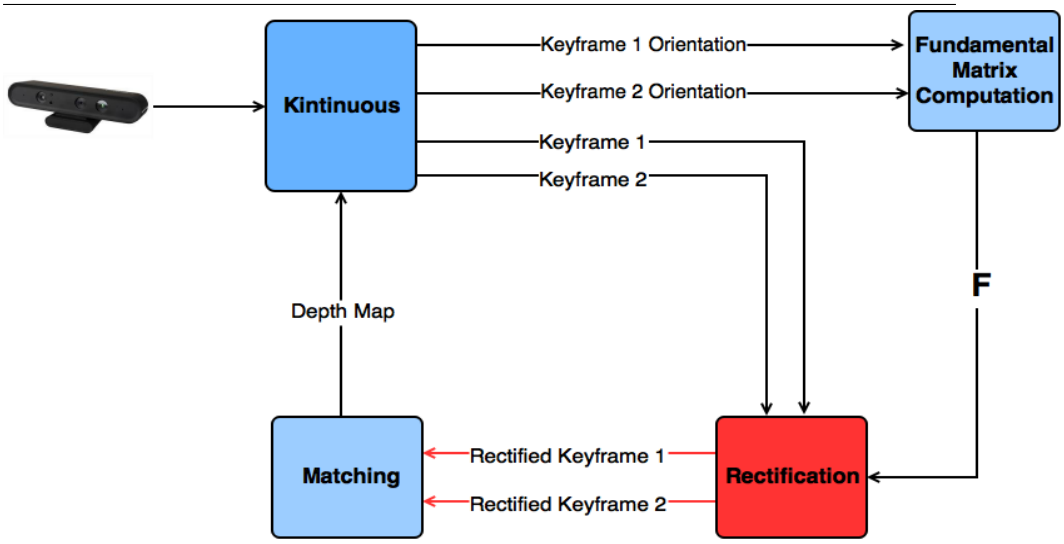
Figure 3.8 The Essential Matrix products relating the Keyframes in geometrical coordinates



3.4 Rectification of the Keyframes

The rectification of the Keyframes is done by the Rectification system component, which is highlighted in figure 3.9.

Figure 3.9 System overview, highlighting the Rectification component

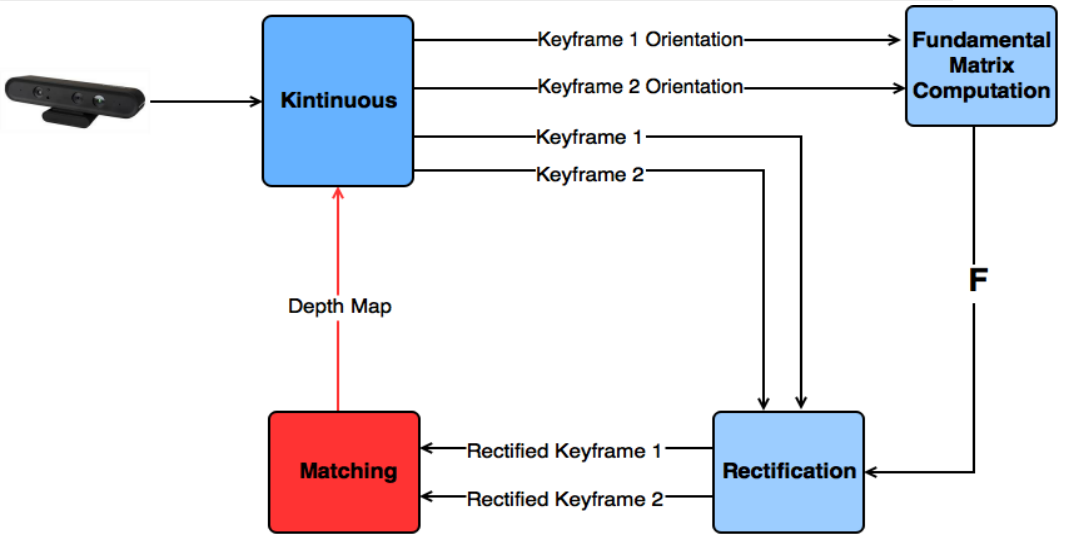


When dealing with general motion, the epipoles may very well lie in the images. When this is the case, traditional rectification schemes fails as the transformation of the image planes so that the corresponding space planes are coinciding will result in infinitely large images. ”A Simple and Efficient Rectification Algorithm for General Motion” (CITE) was chosen as the rectification algorithm for this project, as it efficiently deals with the rectification of the Keyframes regardless of where the epipoles may lie. The implementation of the algorithm is based on an implementation provided by Github user ”nesthorm”.
The rectification algorithm is implemented as a new class in Kintinuous. When the Fundamental Matrix computation is completed, a new object of the rectification algorithm is created. By calling the rectification objects ”Compute()” method, and passing in the Fundamental Matrix, the algorithm will find the extreme epipoles, determine the common epipolar lines, find the minimum distance between the epipolar lines, and lastly create two OpenCV maps each Keyframe. One map for every x-value in the source image, and one map for every y-value in the source image. The maps tell where each pixel, (x_n, y_n), in the unrectified Keyframe go to form the rectified Keyframe. When all four maps have been computed, the Keyframes are undistorted by using the intrinsic and distortion parameters, and rectified by remapping each Keyframe using its respective x-map and y-map.
Thus, the rectification process will warp the Keyframes to make conjugate epipolar lines row-aligned. Because the matching process will operate on the rectified Keyframes, the resulting depth map will consequently also be rectified. The fusion of a rectified depth map is obviously not desired. Therefore, the inverse maps, which can be used to revert the rectified Keyframes back to their unrectified state, are stored for when the matching process is complete.

3.5 Stereo Matching

The computation of the depth map is done by the Matching system component, highlighted in figure 3.10.

Figure 3.10 System overview, highlighting the Matching component



Due to the inaccuracy of the rectified Keyframes, which will be elaborated in section 4.2, the depth map computation proved itself to be the most challenging part of the system by a long shot. The matching algorithm developed uses descriptors to match the pixels due to the inaccurate rectification, and uses ideas from Semi-Global Matching [22] to counter the high potential for bad matches. This is done by enforcing a smoothness constraint on the disparity image. The algorithm consists roughly of the following steps, which will be elaborated shortly:

1. Compute and cache the descriptors for every pixel in both rectified images.
2. Construct a cost volume, consisting of the five lowest cost disparities for every pixel.
3. Aggregate the costs in the cost volume by preferring continuous disparities.
4. Compute the disparity map by choosing the disparity with the lowest aggregated cost.
5. Computing the depth map from the information about the camera intrinsic parameters.
6. Remapping the depth map to its unrectified state.

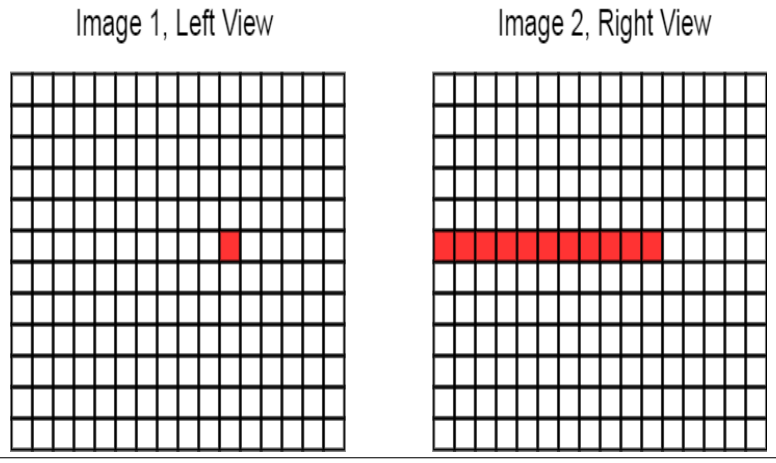
DAISY [23], which is a fast local descriptor for dense matching, was chosen to describe the image pixels due to its speed, highly customizable parameters and performance. While

the computational time of both ORB and BRIEF is lower, DAISY outperformed both descriptors in terms of confidence matches and smoothness of the disparity map. In addition, DAISY descriptors can be computed for every pixel in the image, in contrast to ORB and BRIEF which can not be computed for pixels close to the edges. The decision to use descriptors instead of individual pixel intensities was made because of the inaccuracy of the rectification. Corresponding image points can not be expected to lie on the same row in both images. Searching several rows above and below the current row, not only increases computational time, but also increases the potential for bad matches. DAISY lets you choose the radius of the neighborhood being taken into consideration when computing the descriptors. Setting the radius parameter high enough to counter the inaccuracy of the rectification makes corresponding image row search a possibility.

OpenCV versions higher than 3.0 have an implementation of the DAISY descriptor. Unfortunately, Kintinuous is built around OpenCV 2.4, and does not work with the newer versions. The DAISY source code was therefore downloaded directly from CVLAB.

The matching is done by computing the sum of absolute differences between the pixel's descriptors, where the best match will result in the smallest difference-value. The disparity is the absolute difference between the column number of the pixel in image 1, and the column number of the corresponding pixel in image 2. The difference between the two pixel's descriptors is referred to as the disparity *cost*. Because the matching process should be a quick as possible, the decision was made to use the OpenCV bruteforce matchers on two corresponding rows at a time to decide the corresponding pixels, instead of doing it "manually" for every pixel, due to the OpenCV bruteforce matchers speed. Doing the matching manually is illustrated in figure 3.11. Image 1 is the left view of the scene. The current pixel being operated on is highlighted in red. To find the corresponding pixel, every descriptor of the pixel is highlighted in red in Image 2, which is the right view of the scene, and must be compared to the descriptor of the current pixel in Image 1. The disparity would be the absolute difference between the column number of the pixel in Image 1 and the column number of the pixel in Image 2 that resulted in the lowest cost. Doing this for every pixel in Image 1 to compute the dense disparity map, would take up to several minutes if the rectified images were very large. Needless to say, it is not feasible.

Figure 3.11 Manually matching the descriptors

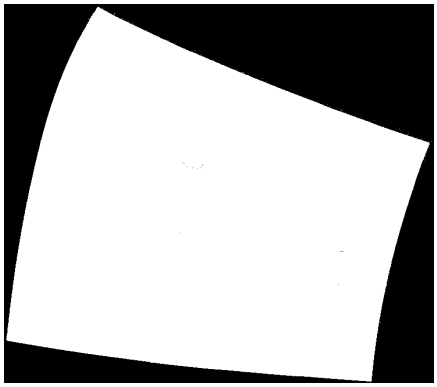


The bruteforce OpenCV matchers takes in two matrices of descriptors and returns a vector of matching objects. Each matching object represents the best match between two descriptors in the descriptor matrices. Each object has three attributes; the query index, the train index, and a distance value. The query index refers to a descriptor in the first descriptor matrix, and the train index refers to a descriptor in the second descriptor matrix. The distance value is the distance between the descriptors (e.g. Hamming distance or Euclidean distance. Depends on the distance calculation chosen). If one of the matching objects has query index value, 4, train index value, 12, and distance value 4.5, it means that descriptor number 4 in the first descriptor matrix best matched with descriptor number 12 in the second descriptor matrix, with the distance between them being 4.5. The OpenCV bruteforce matchers will always return a match for every descriptor in the first matrix. However, this does not necessarily mean that the match found is correct, only that it was the best match of the descriptors provided.

Before performing the matching to find the disparity values, both images are processed by a binary thresholding operation to obtain two masks. Depending on how much the Keyframes are warped during the rectification, large parts of the rectified images do not contain remapped pixels from the unrectified Keyframes. This is especially the case when the epipoles are located on the image planes. The masks will have a white pixel value for pixels that stem from the source images, and a black pixel value otherwise. This is made more clear in figure 3.12. When doing the matching, every pixel is validated beforehand by checking that it is represented by a white pixel in the mask.

Figure 3.12 Rectified Keyframe and its corresponding mask.

(a) Rectified Keyframe

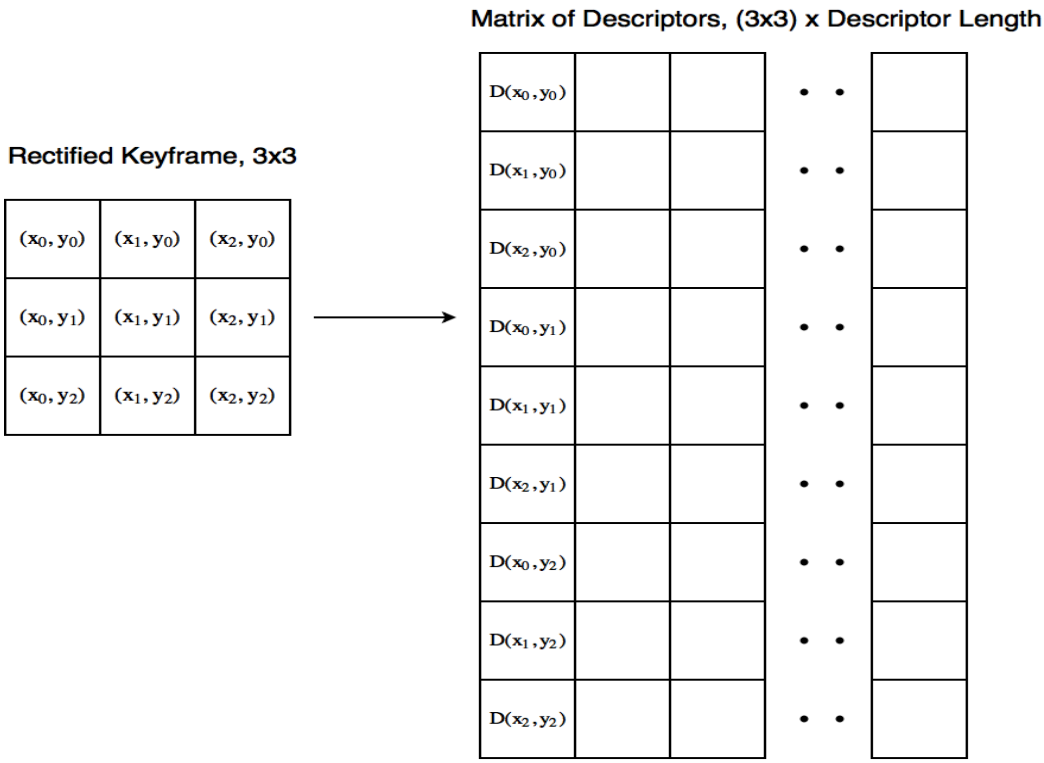


(b) Mask

3.5.1 Computation of the Descriptors

When the rectified Keyframes enter the matching algorithm the DAISY descriptor for every pixel in both images is computed and stored in two matrices of size $(imagewidth * imageheight) \times (descriptorlength)$. The vector describing pixel $(x_n.y_n)$ can then be accessed from row number $x_n + y_n * (imagewidth)$ in the descriptor matrix. This is illustrated in figure 3.13. This provides quick and easy access to every descriptor in both images.

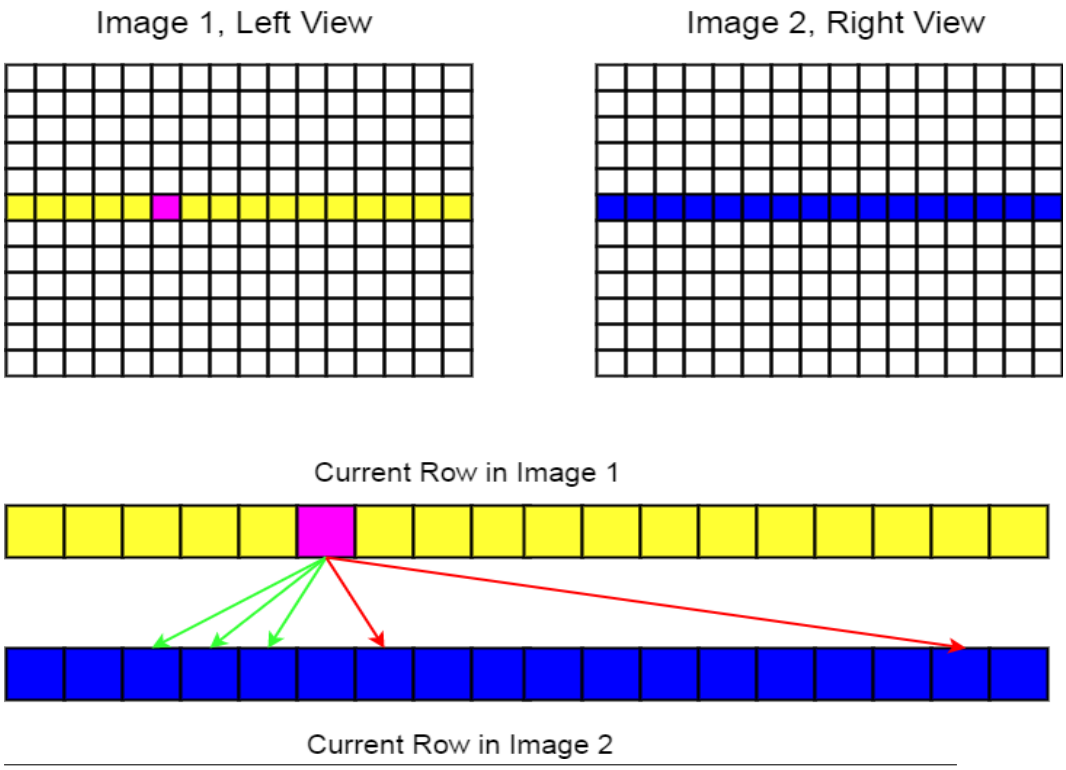
Figure 3.13 Figure illustrating the caching of the descriptors



3.5.2 Construction of the Cost Volume

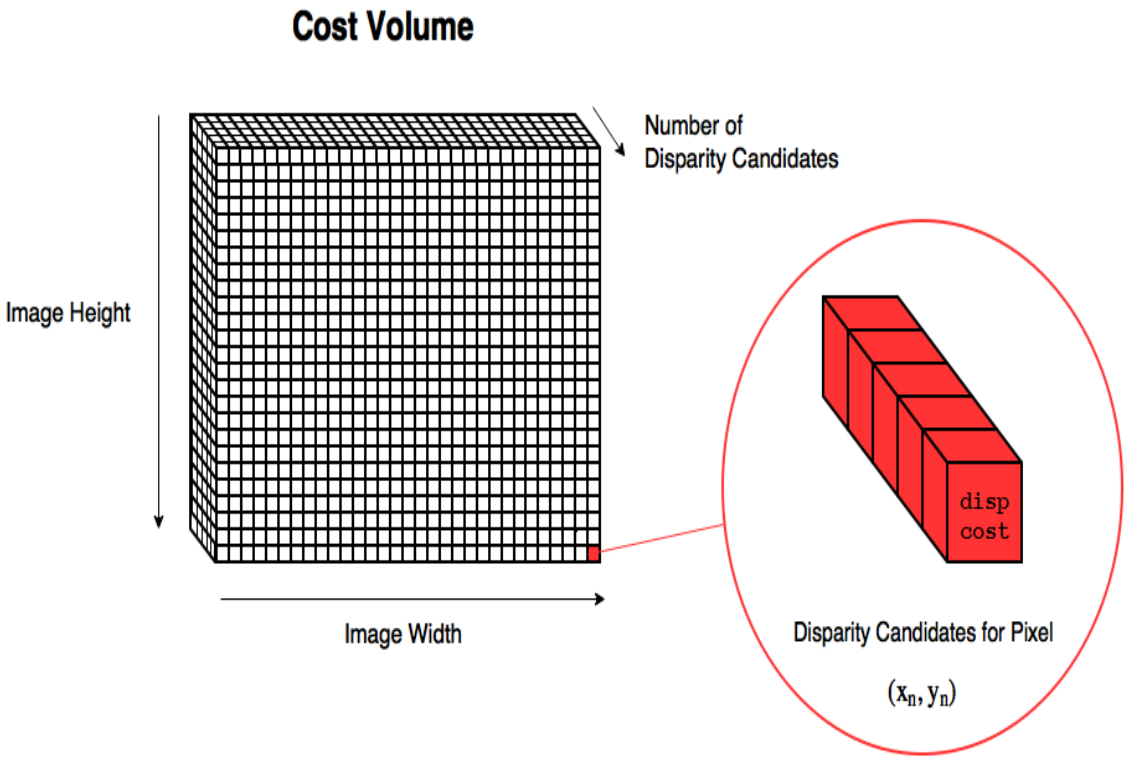
Starting from the top row in rectified Keyframe 1, every valid pixel in the row gets its descriptor copied into a temporary matrix. Then, every valid pixel on the same row in rectified Keyframe 2 gets its descriptor copied into another temporary matrix. The two temporary descriptor matrices are then matched by a 10-nearest neighbour bruteforce matcher. Thus locating the ten best matching pixels on the current row in rectified Keyframe 2, for every pixel on the current row in rectified Keyframe 1. It is not guaranteed that the best match is actually the correct match. Therefore, the ten best matches are found for further processing. The ten best matches for every pixel is then filtered to remove the matches that are surely wrong. Due to knowing the direction of the camera between the capture of each Keyframe, the correspondence of pixel (x_n, y_n) in Keyframe 1 can not lie on both sides of pixel (x'_n, y'_n) in Keyframe 2. Thus, depending on the direction the camera went, impossible correspondences are removed. This is illustrated in figure 3.14. Correspondences that result in a disparity that is larger than the maximum disparity is also removed. The maximum disparity is chosen as the maximum possible disparity value of two corresponding points when an object located 20cm from the sensor. If anything located closer than 20 cm in front of the specific sensor used in the project, the IR-projector will turn off for safety reasons. The maximum possible disparity was found to be 170.

Figure 3.14 The current pixel being filtered is highlighted in purple. The arrows in green and red represent the best matches. The green arrows indicate matches that are accepted. The red arrows are matches that are impossible due to the direction of the camera. These matches are consequently rejected.



To reduce computational time, the five best remaining matches for every pixel are chosen, and stored in a cost volume. The cost volume is of size image rows $\times$ image columns $\times$ 5. Every cell in the cost volume contain two values; the disparity value of the match and its cost. The position of each cell with regards to height and width of the volume, represents the pixel coordinate in rectified Keyframe 1. The cost volume is illustrated in figure 3.15. When the number of matches found is fewer than five, the cells that do not contain known values get a disparity of zero, and a low cost. When the matches are fewer than five, it is usually due to the fact that the row being processed is located at the edge of the image. Therefore it is probably not a part of the coinciding region between the rectified Keyframes.

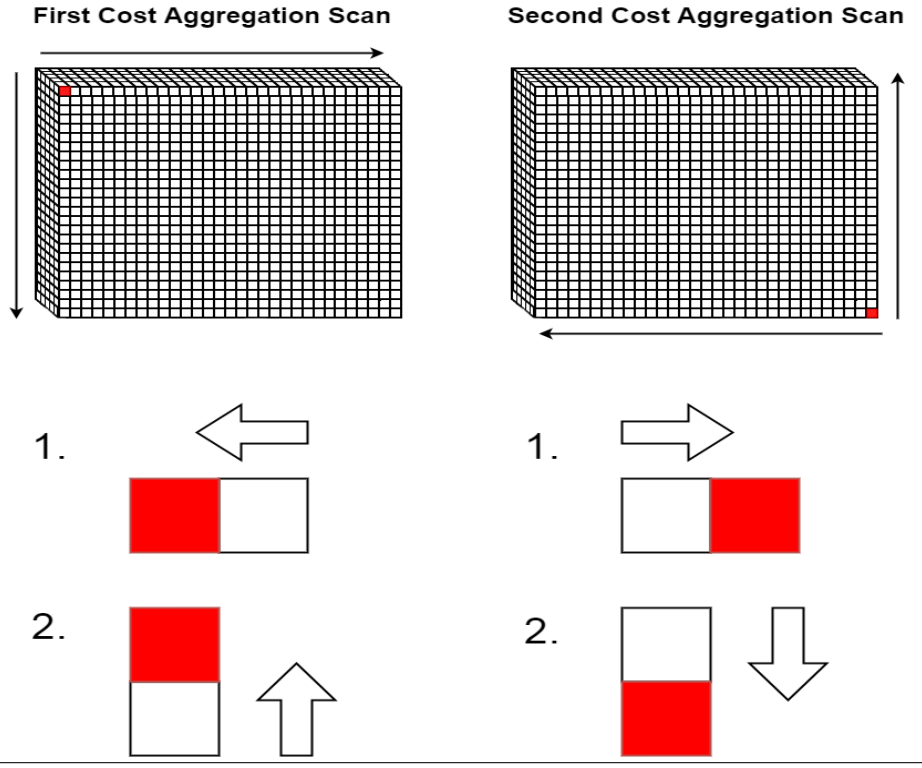
Figure 3.15 The cost volume created. Every cell contains a disparity value and a cost.



3.5.3 Cost Aggregation

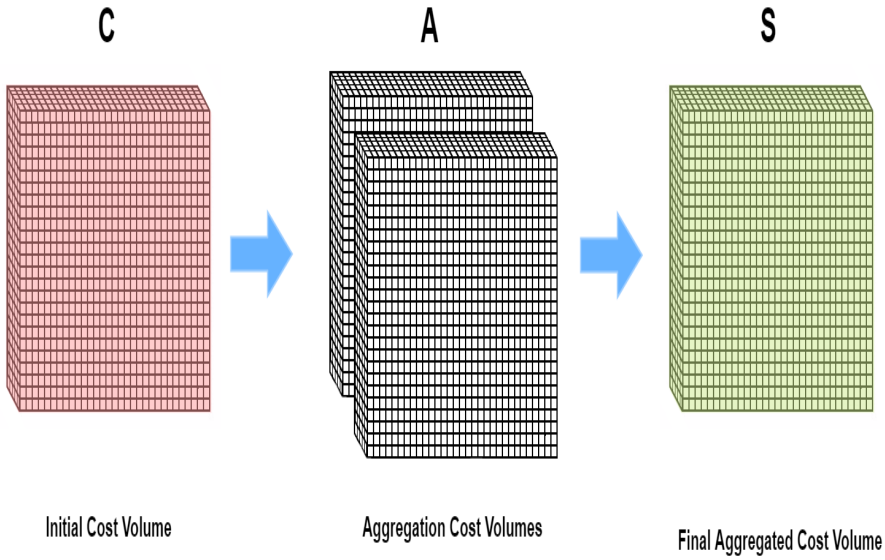
The cost volume contains five disparity candidates, and their respective cost, for every valid pixel in rectified Keyframe 1. A disparity map could now be created by choosing the disparity candidate with the lowest cost for every pixel. However, as stated earlier, the disparity candidate with the lowest cost is not necessarily always the correct match. The cost volume is therefore processed by preferring continuous disparities. This is done by looking in four different directions at each current disparity candidate, one direction at a time. This is illustrated in figure 3.16. Two scans of the cost volume are performed, where each scan looks in two directions.

Figure 3.16 Each cell is compared to its neighbours, penalizing discontinuous disparities with cost penalties. The first scan compares the current cell to every cell to its left and above, starting from the top left corner. The second scan compares the current cell to every cell to its right and below, starting from the bottom right corner. The current cell is colored red, and the cell it is being compared to is colored blue.



This introduces two new cost volumes. The Final cost volume, denoted as S, and the aggregation cost volume, denoted as A. Cost volume S is of size number of candidates $\times$ height $\times$ width, and every cell contains a disparity value and a cost. Cost volume, A, is of size 2 $\times$ candidates $\times$ height $\times$ width, and every cell also contains a disparity value and a cost.

Figure 3.17 Cost volumes C, A, and S.



Every cell in S and A is initialized with zero cost, and the disparity value for each cell is copied from C. How the cost aggregation is performed is explained in the following pseudocode:
First scan:

- **For** every row, starting from row 0 and going down
  - **For** every column, starting from column 0 and going right
    - **For** every disparity candidate, starting from candidate 0
      1. The aggregated cost equals the cost of C[row][col][n]
      2. **If** [row][col-1] exists:
         (a) Create an empty set of costs.
         (b) **For** every disparity candidate at A[1][row][col-1]:
            i. **If** the disparity of this candidate equals the disparity of A[1][row][col][n], the cost of this candidate is added to the set of costs.
            ii. **Else if** the disparity of this candidate equals the disparity of A[1][row][col][n] +-1, the cost of this candidate, plus a small cost penalty, is added to set of costs.
            iii. **Else**, the cost of this candidate, plus a large cost penalty, is added to the set of costs.
         (c) The smallest entry in the set of costs is added to the aggregated cost
      3. **If** [row-1][col] exists:
         (a) Create an empty set of costs.

(b) **For** every disparity candidate at A[1][row-1][col]:

i. **If** the disparity of this candidate equals the disparity of A[1][row][col][n], the cost of this candidate is added to the set of costs.

ii. **Else if** the disparity of this candidate equals the disparity of A[1][row][col][n] +-1, the cost of this candidate, plus a small cost penalty, is added to set of costs.

iii. **Else**, the cost of this candidate, plus a large cost penalty, is added to the set of costs.

(c) The smallest entry in the set of costs is added to the aggregated cost

4. The aggregated cost is added to the cost of A[1][row][col][n].
5. The cost of A[1][row][col][n] is added to the cost of S[row][col][n].

Second scan:

- **For** every row, starting from the number of rows and going up
  - **For** every column, starting from the number of columns and going left
    - **For** every disparity candidate, starting from candidate 0
      1. The aggregated cost equals the cost of C[row][col][n]
      2. **If** [row][col+1] exists:
         (a) Create an empty set of costs.
         (b) **For** every disparity candidate at A[2][row][col+1]:
            i. **If** the disparity of this candidate equals the disparity of A[2][row][col][n], the cost of this candidate is added to the set of costs.
            ii. **Else if** the disparity of this candidate equals the disparity of A[2][row][col][n] +-1, the cost of this candidate, plus a small cost penalty, is added to set of costs.
            iii. **Else** the cost of this candidate, plus a large cost penalty, is added to the set of costs.
         (c) The smallest entry in the set of costs is added to the aggregated cost
      3. **If** [row-1][col] exists:
         (a) Create an empty set of costs.
         (b) **For** every disparity candidate at A[2][row+1][col]:
            i. **If** the disparity of this candidate equals the disparity of A[2][row][col][n], the cost of this candidate is added to the set of costs.
            ii. **Else if** the disparity of this candidate equals the disparity of A[2][row][col][n] +-1, the cost of this candidate, plus a small cost penalty, is added to set of costs.
            iii. **Else**, the cost of this candidate, plus a large cost penalty, is added to the set of costs.
         (c) The smallest entry in the set of costs is added to the aggregated cost
      4. The aggregated cost is added to the cost of A[2][row][col][n].
      5. The cost of A[2][row][col][n] is added to the cost of S[row][col][n].

Thus the final cost of each disparity candidate in cost volume S will be determined by the continuity of the disparity values above, below, to the left and right of each initial disparity candidate. If the disparity of a candidate with low cost is surrounded by candidates with equal disparities, it will be penalized with a large cost for all directions.

3.5.4 Computation of the Disparity and Depth Maps

The computation of the disparity map is done by first creating a new image of the same size as the rectified Keyframes, initializing every pixel to disparity value 0. Then, starting from pixel (x_0, y_0), each pixel is given a disparity value from cost volume S. Pixel (x_n, y_n) in the disparity map is given by the disparity candidate at S[y_n][x_n] that has the lowest cost. If pixel (x_n, y_n) in the mask of rectified Keyframe 1 is represented by a zero-value, pixel (x_n, y_n) will be ignored, and keep its zero disparity value. After the computation of the disparity map is completed, it is converted to a depth map. Recalling that the depth can be determined by the disparity measurement when knowing the camera intrinsic parameters and the baseline:

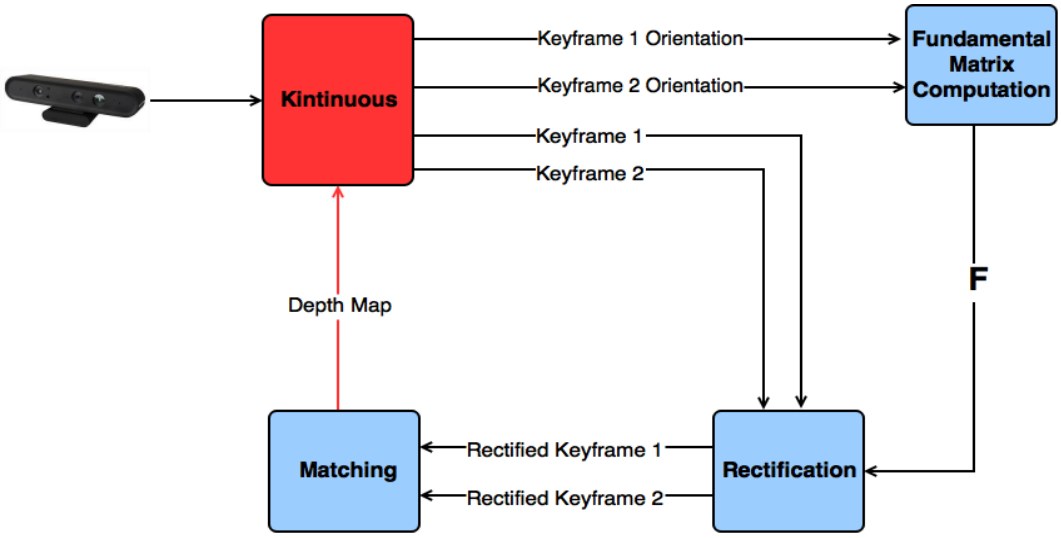
$$Z = \frac{f * T}{d - (c_x - c_y)} \tag{3.7}$$

The depth map is thus acquired by performing this calculation on every disparity value in the disparity map. Finally, the inverse maps provided by the rectification system component is used to remap the depth map to its unrectified state.

3.6 Depth Map Fusing

The depth map fusing component is highlighted in figure 3.18. Unfortunately, it was never actually implemented. This section will briefly explain two of the approaches planned for the fusing of the depth maps.

Figure 3.18 System overview, highlighting the fusing component



3.6.1 Fusing Through the Existing Depth Map Pipeline

Every time Kintinuous receives a depth map and RGB-image from the Kinect they get processed by a class called ”LogReader”, and the data sent to the main class in Kintinuous, the ”KintinuousTracker” class. This is the class that creates and update the dense pose graph, containing all the rotations and translation of the sensor. It takes the current RGB, depth, translation and rotation data, the current timestamp, among other paramteres, and integrates it with the TSDF. This is the main RGB-D pipeline in Kintinuous. If the use of this pipeline would be possible, by sending in the depth map, unrectified Keyframe 1, the translation and rotation of the Keyframe with respect to the world coordinate system, the depth map fusing would be very simple to integrate.

3.6.2 Point Cloud Fusing

If using the main pipeline was unsuccessful, it would might be possible to create a point cloud directly from the disparity map, instead for a depth map, and the fuse this point cloud with the full point cloud being created by Kintinuous. The creation of a point cloud from the disparity map is very straight forward. The x and y position of every point in the point cloud is given by the coordinate of the disparity value, pluss c_x and c_y, respectively. The z-coordinate is found in the same way as when computing the depth map. How to actually do the fusion of the point clouds might have proven itself a bit more tricky, but probably possible.

CHAPTER 4

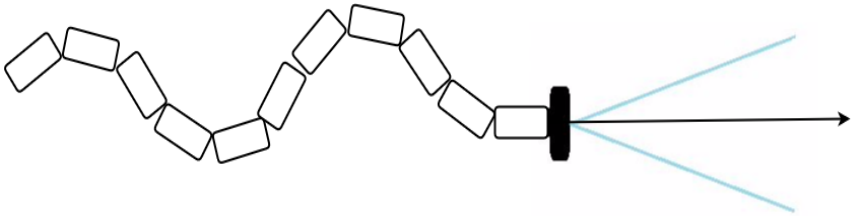
Testing and Results

This chapter provides experiments conducted on the various parts of the system, and the experiments results. Section 4.1 provides experiments conducted and results for the Fundamental Matrix, section 4.2 the experiments conducted and results for the rectification. section 4.3 will give an overview of a select few of the matching algorithm approaches tried, explaining the final matching algorithm in section 4.4, and its results.

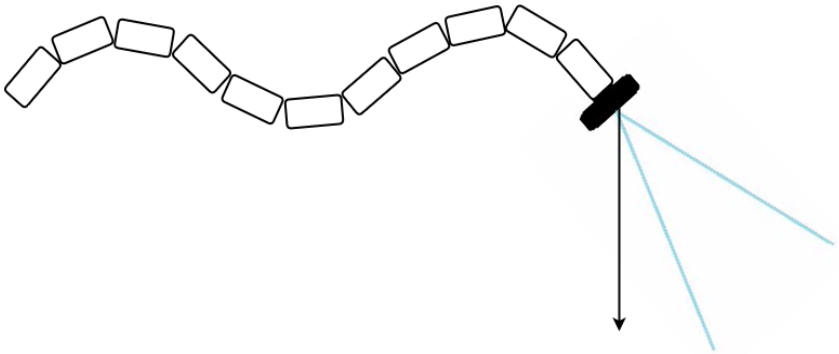
4.1 The Fundamental Matrix

The Fundamental Matrix was tested by conducting one check, and two experiments, per main movement of the camera, with the main movements being forward/backwards and left/right motion. While backward and forward motion coherently being the opposite of each other, the epipolar geometry between the frames are similar in regards to the epipoles. Recalling from section 2.3.2, the epipoles are located where the baseline intercepts the image planes. This should in theory be in the exact same place, as long as the camera moves along the same directional vector, during forward and backwards motion. The same goes for left and right motion, with the exception that the epipoles in this scenario should lie outside of the image planes. These camera movements where chosen from the obvious reason that the head of the snake will move either forward/backwards or left/right.

Figure 4.1 Motion of the camera mounted on the head of the snake



(a) Forward motion



(b) Right motion

The following check and experiments were used in the assessment of the Fundamental Matrix:

1. Computation of the Rank and determinant of the matrix.
2. Computing the distance between the epipolar lines and the points that should lie on them, to see how accurate the matrix is.
3. Comparison to a Fundamental Matrix computed by the 8-point method.

Computing the Rank and Determinant Reasoning

Computing the rank and determinant of the Fundamental Matrix is an obvious first check that doesn't say anything about the accuracy of the matrix, but at least confirms that it has the properties of a Fundamental Matrix.

Distance Comutation Reasoning

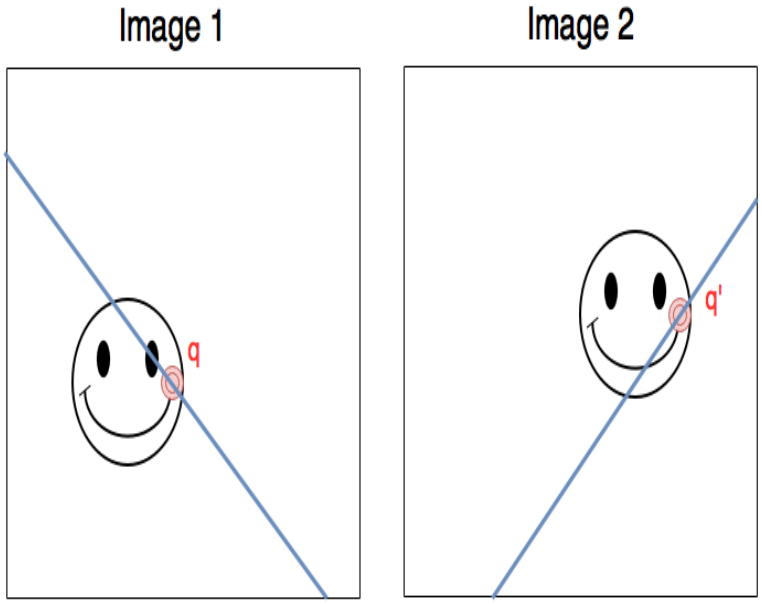
To give merit to the second experiment, consider a perfectly accurate Fundamental Matrix relating a stereo image pair. The Fundamental Matrix relates two corresponding points in

the image pair, q and q', with the following equation:

$$q^T F q' = 0 \tag{4.1}$$

The conjugate epipolar lines given by the equation will for this perfectly accurate Fundamental Matrix lie exactly on top of the corresponding points, as illustrated in figure 4.2. How accurate the Fundamental Matrix computed from Kintinuous actually is will be reflected by the distances between the point correspondences and the epipolar lines.

Figure 4.2 Two corresponding points in a stereo image pair. The perfectly accurate Fundamental Matrix makes the conjugate epipolar lines, drawn in blue, lie exactly on top of its corresponding point.



For the execution of this experiment, an OpenCV feature detector was used to find strong features in both images. The features were then matched and filtered with a cross-check to obtain only high confidence point correspondences between the images. The points were then multiplied with the Fundamental Matrix, resulting in epipolar lines in Keyframe 1 and Keyframe 2, respectively. The distances in pixels from the points to the epipolar lines were then written to a file and plotted in Matlab.

Comparison to the 8-point Method Reasoning

The third test consisted of comparing the Fundamental Matrix from Kintinuous to a Fundamental Matrix obtained by the 8-point method. OpenCV's 8-point method algorithm was utilized to compute the matrix using the features found by the previous experiment. The features **not** used in the computation of the matrix were then spent computing the epipolar lines. The distances in pixels between the epipolar lines and the points were then

written to a file, plotted in Matlab, and compared to the Kintinuous results.
The reasoning behind the choice of this experiment was to see how the Kintinuous Fundamental Matrix compared to the 8-point method matrix in terms of accuracy. By having access to the orientation of the camera when the Keyframes are taken the computation of the Fundamental Matrix is very quick. In addition, a Fundamental Matrix is guaranteed, in contrast to the 8-point method, where good features have to be present in both images. However, if the accuracy of the 8-point method matrix outclasses the one from Kintinuous, this might give us reason to consider choosing the slower approach.

Execution of the Experiments and Motivation

The experiments were done by holding the Kinect sensor by hand and moving it either left, right, backwards or forwards, emulating the movement of the snakehead. The epipolar lines were drawn directly on top of the Keyframes, conjugate epipolar lines being in the same color, along with circles indicating the point used to calculate its conjugate epipolar line. This was done using the Fundamental Matrix from both Kintinuous and the 8-point method in the same run, using the same point correspondences to draw the epipolar lines. All tests resulted in the Kintinuous Fundamental Matrix being of rank 2, with determinant either equal to 0, or some small value (10^{-25}).

The Stereo matching performance depends on the rectification of the Keyframes , which in turn depends on how accurate the Fundamental Matrix is. Therefore, a thorough assessment of the Fundamental Matrix is important as its accuracy has a direct impact on the accuracy of the rectification.

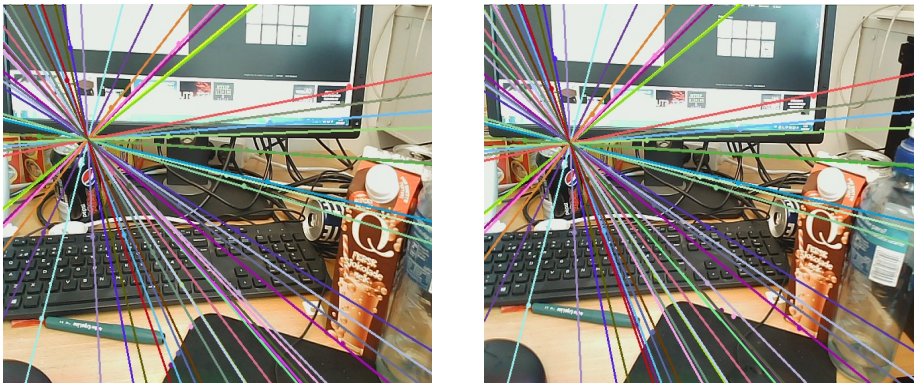
4.1.1 Forward and Backward Motion

Figures 4.3 and 4.4 shows the Keyframes from forward and backward motion of the camera, respectively. The epipolar lines are computed from Kintinuous' Fundamental Matrix and drawn directly on the images. During forward and backward motion the epipoles are expected to lie on the image planes, and, depending on the strictness of the motion, in approximately the same spot in both keyframes. Judging by the figures, it is clear that this is the case.

Figure 4.3 Fundamental Matrix from Kintinuous during **forward** motion. Conjugate epipolar lines and point correspondences are drawn in the same colors.

(a) Keyframe 1 **(b)** Keyframe 2

Figure 4.4 Fundamental Matrix from Kintinuous during **backward** motion. Conjugate epipolar lines and point correspondences are drawn in the same colors.

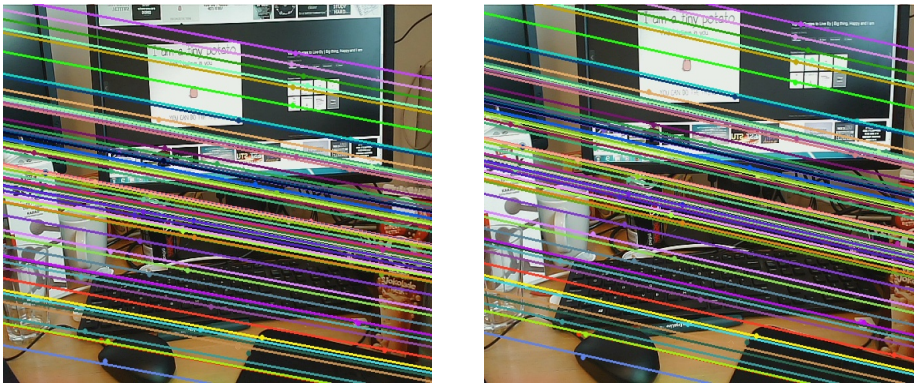


(a) Keyframe 1 **(b)** Keyframe 2

Figures 4.5 and 4.6 shows the Keyframes from forward and backwards motion with the epipolar lines computed from the 8-point algorithm Fundamental Matrix. The epipoles

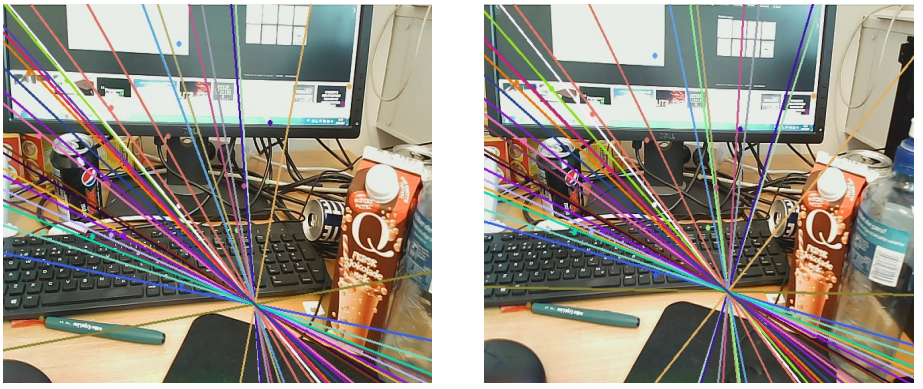
are way off to the sides of each Keyframe during forward motion, which points to a wrong Fundamental Matrix estimation. Similarly to the Kintinuous Fundamental Matrix, the epipoles are located on the image planes during backwards motion, which is to be expected.

Figure 4.5 Fundamental Matrix from 8-point method during forward motion. Conjugate epipolar lines and point correspondences are drawn in the same colors.



(a) Keyframe 1 **(b)** Keyframe 2

Figure 4.6 Fundamental Matrix from 8-point method during backward motion. Conjugate epipolar lines and point correspondences are drawn in the same colors.

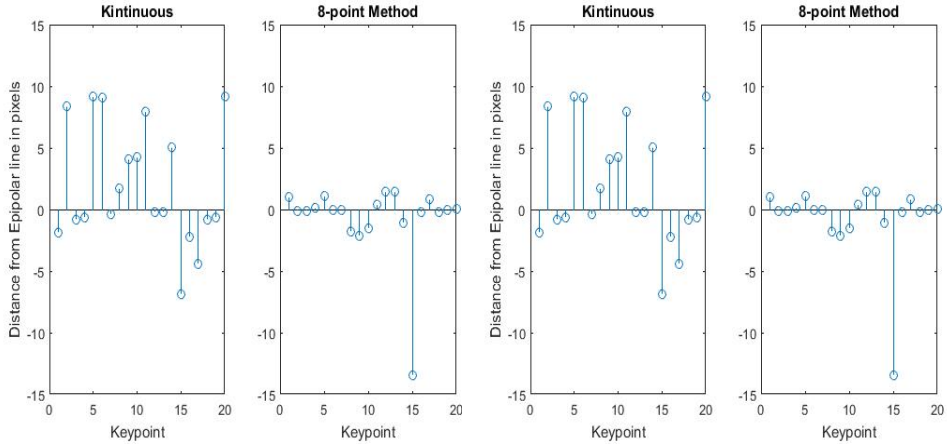


(a) Keyframe 1 **(b)** Keyframe 2

The stem plots in figures 4.7 and 4.8 compare the distances between the points and the epipolar lines for the Kintinuous and 8-point method Fundamental matrices for forward and backward motion, respectively. Figure 4.7 shows that the Fundamental Matrix from the 8-point algorithm has much better accuracy in terms of distance from the keypoints to the epipolar lines. However, the epipoles are expected to lie on the image planes during

forward motion. Judging by the keyframes in figure 4.5 the epipoles are located nowhere near the image planes. Thus, the Fundamental Matrix estimated by the 8-point algorithm fits quite well with the point correspondences, but is most likely not representing an accurate geometrical relation between the keyframes. For backward motion, the 8-point method looks to find a more correct geometrical relation between the keyframes, as the epipoles are in fact on the image planes, as seen in figure 4.6. However, the accuracy is much worse than the Fundamental Matrix from Kintinuous, judging by the stem plot in figure 4.8.

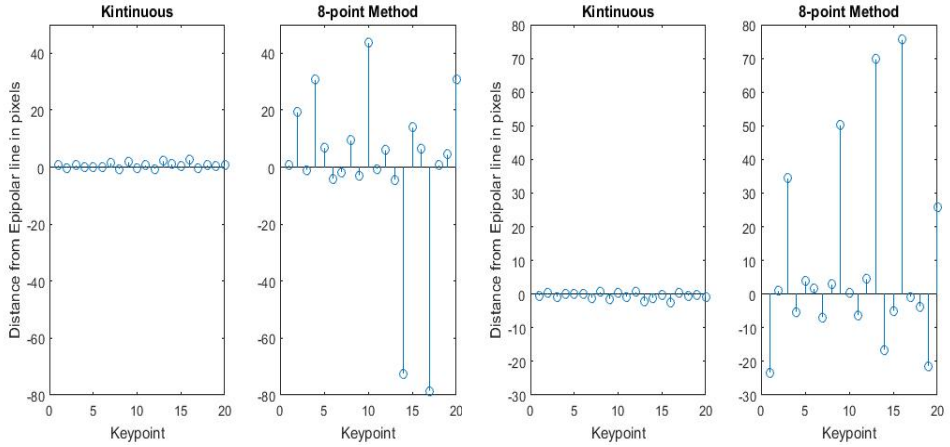
Figure 4.7 Stem plot comparing the distance in pixels from the Keypoints to the epipolar lines, using Kintinuous and 8-point method Fundamental Matrices during forward motion.



(a) Keyframe 1, forward motion

(b) Keyframe 2, forward motion

Figure 4.8 Stem plot comparing the distance in pixels from the Keypoints to the epipolar lines, using Kintinuous and 8-point method Fundamental Matrices during backward motion.



(a) Keyframe 1, backward motion **(b)** Keyframe 2, backward motion

Table 4.1 shows the minimum, maximum, average and median distance in pixels from the keypoints to the epipolar lines during forward motion. While the 8-point algorithm outclasses the Kintinuous Fundamental Matrix in terms of average and median distances, surprisingly enough the maximum distance is much higher than the one from Kintinuous. Again, this is probably due to the Fundamental Matrix actually being a wrong relation between the images, resulting in many small distances, but some large distances that does not fit the matrix. On the other side of the table, the Kintinuous Fundamental Matrix struggles with a quite large average and median deviation from the epipolar lines. The maximum distance is lower than the 8-point algorithm, but still problematically large.

Table 4.1: Distance Statistics of Forward Motion Experiment

Distance From Points to Epipolar Lines (unit is pixels)				
Image	**Min**	**Max**	**Avarage**	**Median**
Kintinuous Image 1	0.19903	9.30054	4.414	4.274
Kintinuous Image 2	0.259827	10.115	4.727	4.608
8-Point Image 1	0.015	13.46	0.8954	0.539
8-Point Image 2	0.01656	14.67	0.9795	0.5886

Table 4.1 shows the minimum, maximum, average and median distance in pixels from the keypoints to the epipolar lines during backward motion. In this case, the Fundamental Matrix from Kintinuous shows near perfect behaviour, with the average and median distance being below one pixel off the epipolar lines. However, the maximum distance is above two pixels, which suggest that it is not perfect, and this imperfection will affect

the rectification. The Fundamental Matrix from the 8-point method struggles quite a bit, which is reflected by the high average and minimum distances, as seen in the table.

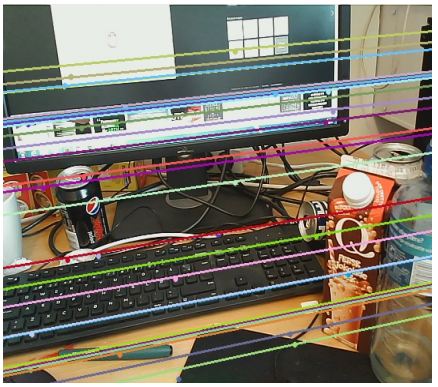
Table 4.2: Distance Statistics of Backward Motion Experiment

Distance From Points to Epipolar Lines (unit is pixels)				
Image	**Min**	**Max**	**Avarage**	**Median**
Kintinuous Image 1	0.04	2.53	0.79	0.69
Kintinuous Image 2	0.03	2.34	0.75	0.63
8-Point Image 1	0.30	78.6	15.8 4	6.36
8-Point Image 2	0.31	75.7	16.4	5.47

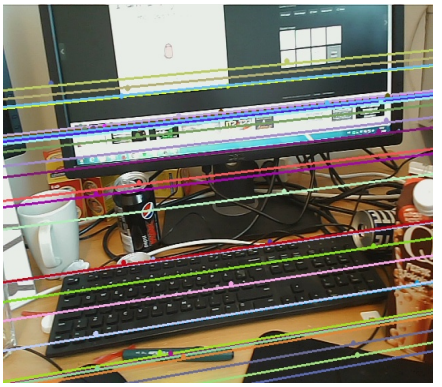
4.1.2 Left and Right Motion

Figures 4.9 and 4.10 shows the Keyframes, with conjugate epipolar lines computed from the Fundamental Matrix from Kintinuous, during left and right motion. As expected, the epipoles are located somewhere far off from the image planes, due to the close-to parallel epipolar lines.

Figure 4.9 Fundamental Matrix from Kintinuous during left motion. Conjugate epipolar lines and point correspondences are drawn in the same colors.

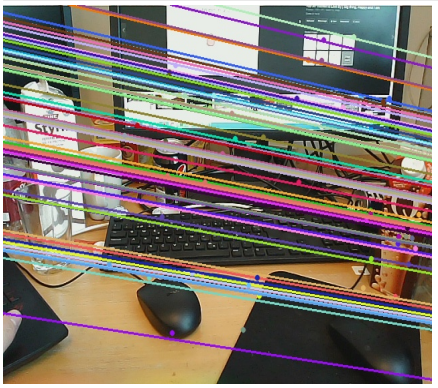


(a) Keyframe 1

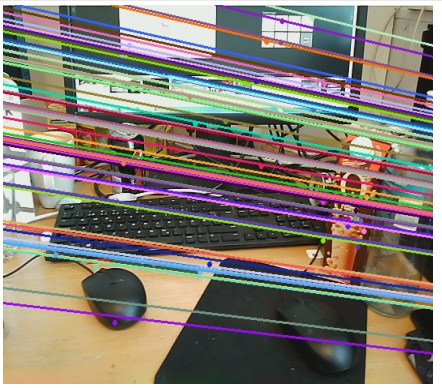


(b) Keyframe 2

Figure 4.10 Fundamental Matrix from Kintinuous during right motion. Conjugate epipolar lines and point correspondences are drawn in the same colors.



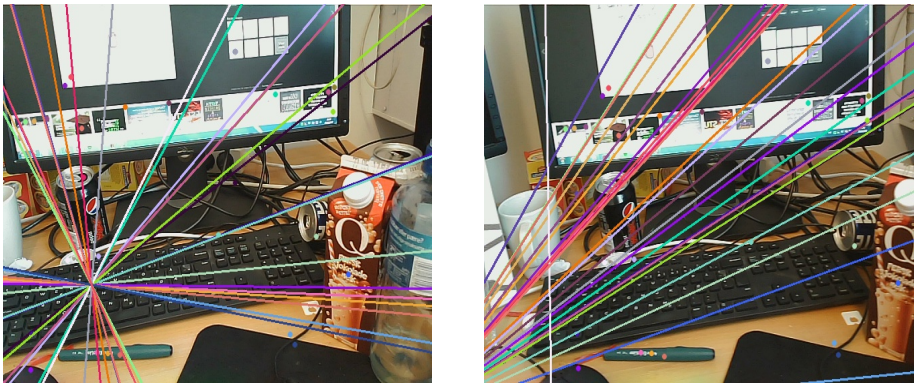
(a) Keyframe 1



(b) Keyframe 2

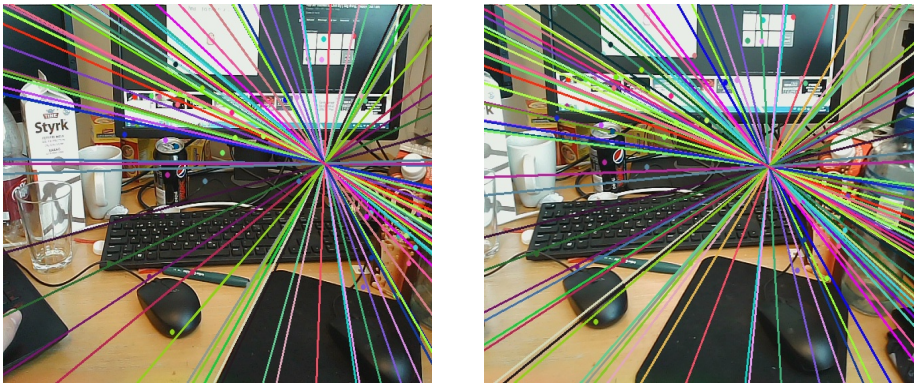
Figures 4.11 and 4.12 shows the Keyframes and their conjugate epipolar lines computed from the 8-point method, during left and right motion. Oddly enough, the epipoles are located on the image planes, which is what would have been expected from forward/backward motion, and not left/right.

Figure 4.11 Fundamental Matrix from 8-point method during left motion. Conjugate epipolar lines and point correspondences are drawn in the same colors.



(a) Keyframe 1 **(b)** Keyframe 2

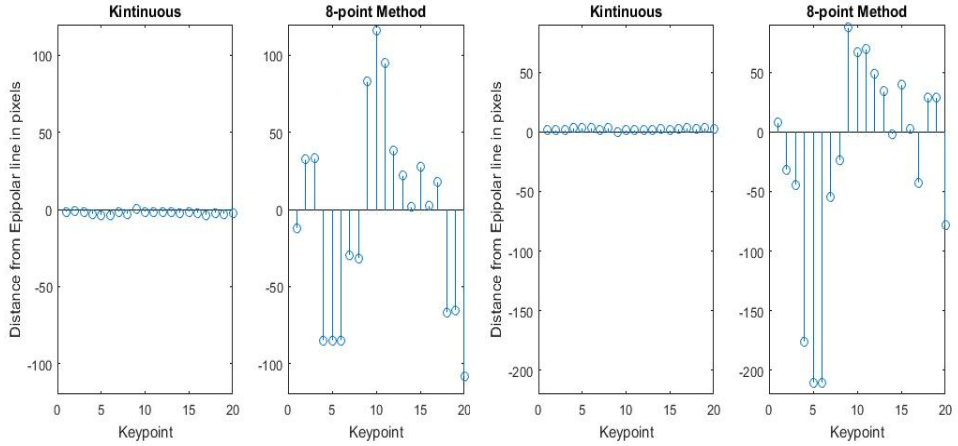
Figure 4.12 Fundamental Matrix from 8-point method during right motion. Conjugate epipolar lines and point correspondences are drawn in the same colors.



(a) Keyframe 1 **(b)** Keyframe 2

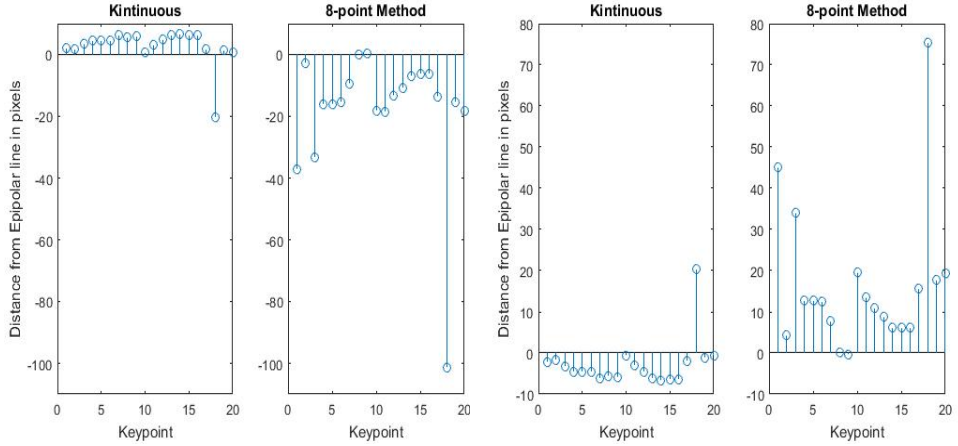
Figures 4.29 and 4.33 show the stem plot showing the distances from the epipolar lines to the points, during left and right motion. The 8-point method Fundamental Matrix suffers greatly from a bad estimation of the geometric relation between the images, which is reflected by the many outliers. The Kintinuous Fundamental Matrix performs better, but is far from perfect.

Figure 4.13 Stem plot comparing the distance in pixels from the Keypoints to the epipolar lines, using Kintinuous and 8-point method Fundamental Matrices during left motion.



(a) Keyframe 1, left motion **(b)** Keyframe 2, left motion

Figure 4.14 Stem plot comparing the distance in pixels from the Keypoints to the epipolar lines, using Kintinuous and 8-point method Fundamental Matrices during right motion.



(a) Keyframe 1, right motion **(b)** Keyframe 2, right motion

Tables 4.3 and 4.4 shows the minimum, maximum, average and median distances from the epipolar lines to the point for left and right motion.

Table 4.3: Distance Statistics of Left Motion Experiment

Distance From Points to Epipolar Lines (unit is pixels)				
Image	**Min**	**Max**	**Average**	**Median**
Kintinuous Image 1	0.65	3.78	2.32	2.16
Kintinuous Image 2	0.63	3.72	2.28	2.10
8-Point Image 1	2.68	187	63	50
8-Point Image 2	1.99	291	78.3	49.7

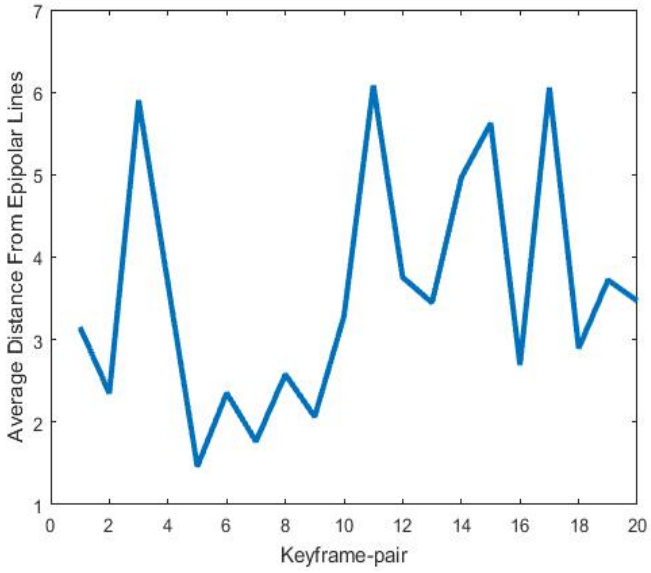
Table 4.4: Distance Statistics of Right Motion Experiment

Distance From Points to Epipolar Lines (unit is pixels)				
Image	**Min**	**Max**	**Average**	**Median**
Kintinuous Image 1	0.03	20.4	3.56	2.87
Kintinuous Image 2	0.03	20.4	3.73	2.89
8-Point Image 1	0.145	101	16.7	16.38
8-Point Image 2	0.12	75	17	17.9

4.1.3 Prolonged Run Average Distances

The accuracy of the Fundamental Matrix will, as stated earlier, affect the rectification, which will in turn affect the challenge of the stereo matching. Thus, a final experiment was conducted to find out how variable the accuracy of the Fundamental Matrix is when using the tracking information from Kintinuous. This experiment consisted of letting the distance calculation algorithm run, and calculating the average distances from the epipolar lines to the point matches, while moving the camera around in all directions. The result can be seen in figure 4.15.

Figure 4.15 The average distances from the points to the epipolar lines, when moving the camera around in all directions.



The average of all the distances from the 20 individual Keyframe pairs was found to be 3.5 pixels. This boils down to the assumption that the rectification will probably have a 3.5 pixel error, which should be taken into account when doing the stereo matching.

4.1.4 The Accuracy of the Fundamental Matrix

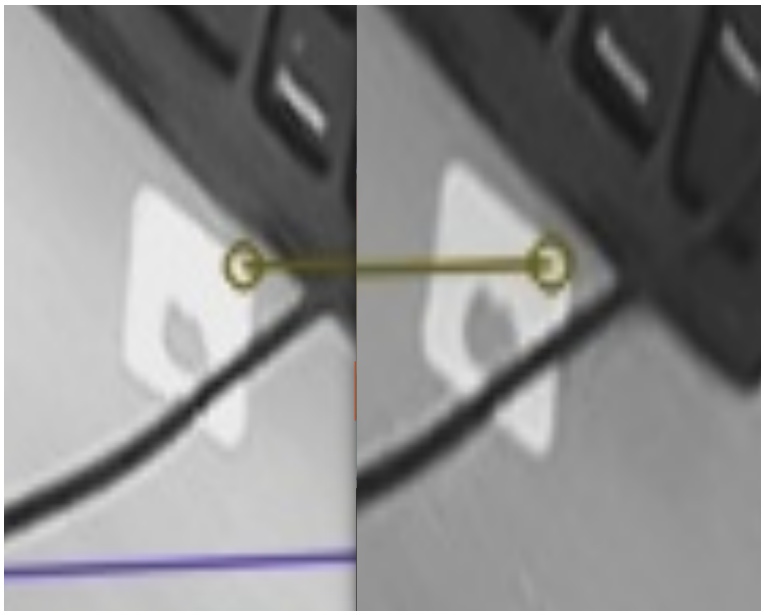
While the accuracy of the Fundamental Matrix from Kintinuous in terms of distance between the points and the epipolar lines is not great, the geometrical relation between the keyframes seems consistently accurate, judging by the epipoles, in contrast to the 8-point method Fundamental Matrix. There are many probable reasons for why the points are deviating from the epipolar lines:

- Inaccurate data from Kintinuous
- Distortion of the images
- Inaccuracy of the OpenCV feature detectors

The most obvious probable cause is that the rotation matrix and translation vector relating the keyframes computed from Kintinuous' orientation data is not completely accurate, and thus affects the Fundamental Matrix. Distortion of the images might also affect the distances between the epipolar lines and point correspondances. Lens distortion results in a warping of the objects and surroundings in the images. This leads to a deviation between the scene being photographed, and the actual image representation of the scene. The Fundamental Matrix inherits no information about the distortion parameters, and thus

some inaccuracy might come from image distortion.
To find the keypoints in the images, an OpenCV keypoint extractor was utilized. A small error, but and error nonetheless, in the distance between the keypoints and the epipolar lines might come from the possibility that the corresponding points do not lie exactly on on the correct pixels. Consider figure 4.16 showing a matched feature correspondence by an OpenCV matcher:

Figure 4.16 Features matched by an OpenCV matcher.



While this obviously is considered a good match, it is easy to see by visual inspection that the feature does not come from exactly the same spot in both images. These few pixel's difference will clearly have an impact on the distance from the keypoints to the epipolar lines whenever a match like this occurs.

To address the bad performance of the 8-point method Fundamental Matrices, this is most likely due to only using 8 points to compute the matrix. By including more point matches, the estimated matrix would perform better, and probably outperform the Fundamental Matrix from Kintinuous in terms of accuracy. But the fact of the matter is, depending on the scene being captured there might often be scenarios where the computed point matches are close to 8, or even fewer. When this is the case, the 8 point method will perform badly, or even fail. Thus, computing the Fundamental Matrix from Kintinuous is the safer choice in terms of consistency.

4.2 Rectification

The assessment of the rectification was done in a similar fashion as the assessment of the Fundamental Matrix. After computing the rectified images, an OpenCV feature detector was used to find keypoints in both rectified images. The keypoints were then matched and filtered to only obtain high confidence matches. For a perfectly rectified image pair, the corresponding points will lie on the the same row in both images, as each conjugate row in the images will be conjugate epipolar lines.

Figure 4.17 Point matches between a perfectly rectified image pair.

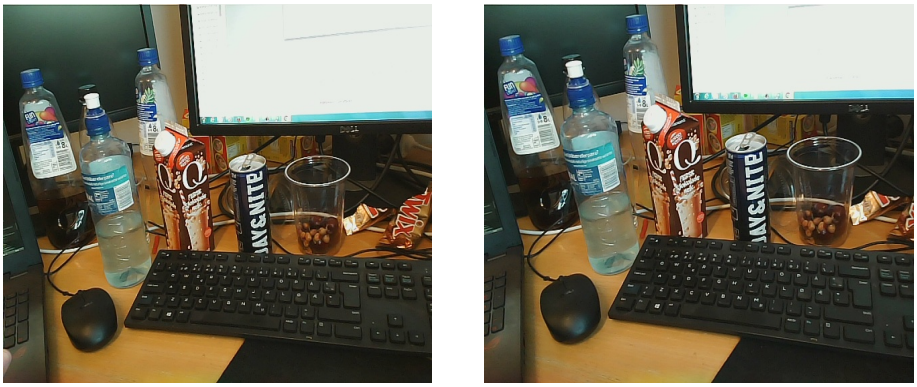
Figure 4.17 shows a stereo image pair from the Middelbur dataset, which is perfectly rectified. Clearly, every corresponding point match is located on the same row in both images. This procedure was done for forward, left, right and backward motion. As forward and backward motion will cause a much more drastic warping of the images, seeing how this affects the accuracy of the rectification is desired. If the drastic warping of the images during the rectification causes the accuracy to be worse, this should be taken into account when doing the stereo matching for forward and backward rectified images.

4.2.1 Forward and Backward Motion

Forward Motion

The following keyframes were obtained, undistorted, and passed into the rectification algorithm. The Keyframes are shown in figure 4.18 and the rectified Keyframes in figure 4.19.

Figure 4.18 Two keyframes obtained by forward motion of the camera.



(a) Keyframe 1 **(b)** Keyframe 2

Figure 4.19 The rectified keyframes obtained by forward motion of the camera

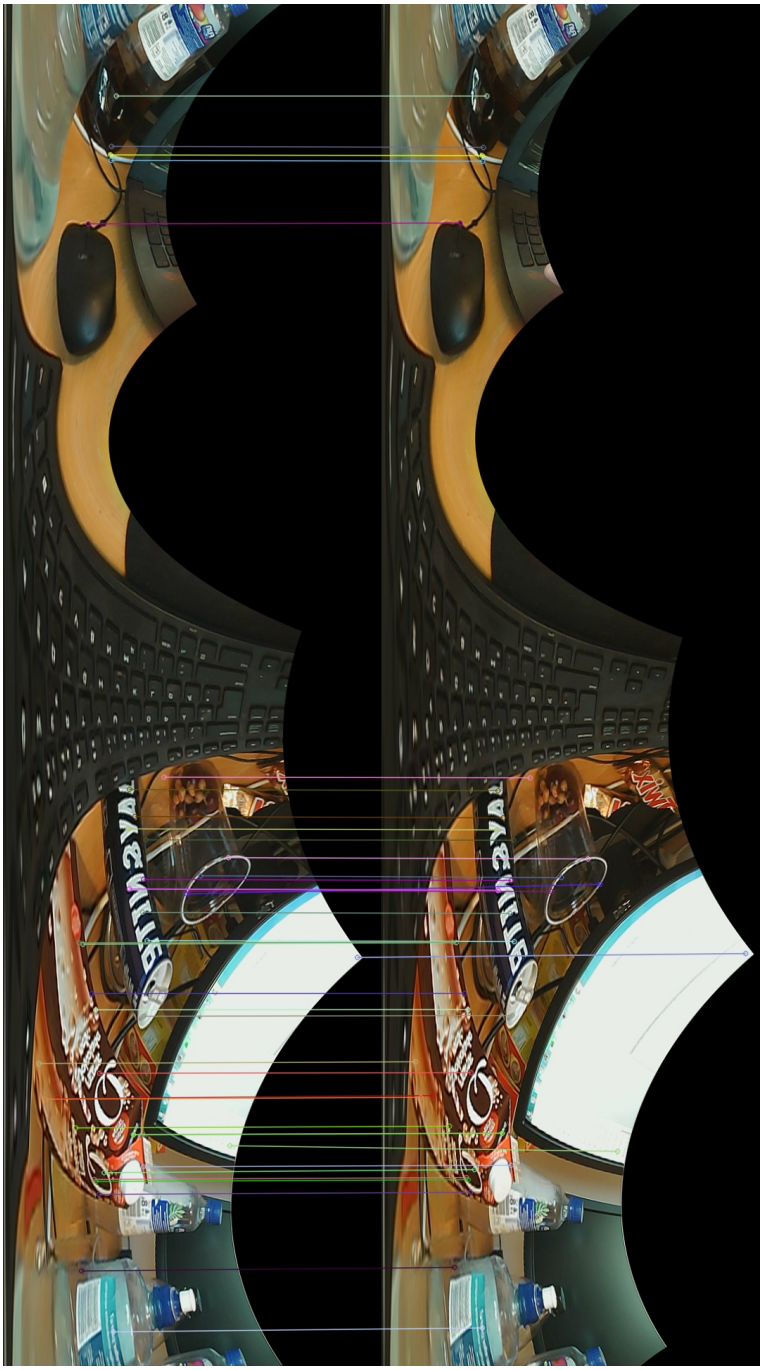
(a) Rectified Keyframe 1 **(b)** Rectified Keyframe 2

The epipoles are located at the left part of the keyboard, due to the aggressive warping

around these points. The time in took the algorithm to perform the rectification was in this case 16.62ms.

Figure 4.20 shows the corresponding points found by the OpenCV descriptor matcher. Due to the aggressive warping of the images, the OpenCV descriptor matcher struggled with finding good correspondences, and even strict filtering of the matches found resulted often in many bad matches. This is troublesome, as it predicts challenges when doing the stereo matching.

Figure 4.20 Corresponding points between the rectified keyframes



The plot shown in figure 4.21 provides the difference in rows between each corresponding point, that should be lying on the same row. Table 4.5 provides the minimum, maximum, average and median difference in rows. While the maximum difference is quite large, the average and median values are relatively low. A very useful discovery was made when doing this experiment; when the images are rectified during forward motion, the corresponding points in rectified Keyframe 1 always lies to the left their conjugate point in rectified Keyframe 2. Similarly to left and right motion, this constrains the search for pixel matches when doing the stereo matching.

Figure 4.21 The difference in rows between corresponding points in rectified image pair

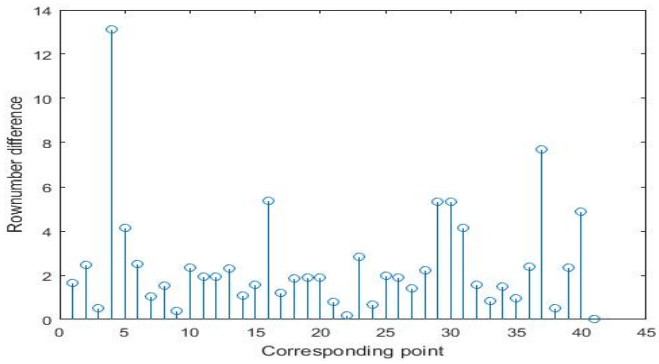


Table 4.5: Distance Statistics of Rectified Image pair

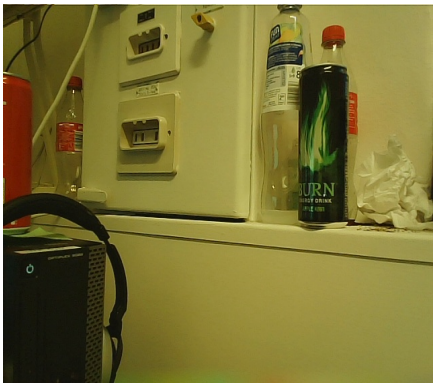
Difference in Rows Between Corresponding Matches (units are pixels)			
Min	**Max**	**Average**	**Median**
0.0014	13.1	2.444	1.893

Backward Motion

Figures 4.22 and 4.23 shows the Keyframes obtained from backward motion and their rectifications, respectively. It took the algorithm 16.73 ms to perform the rectification.

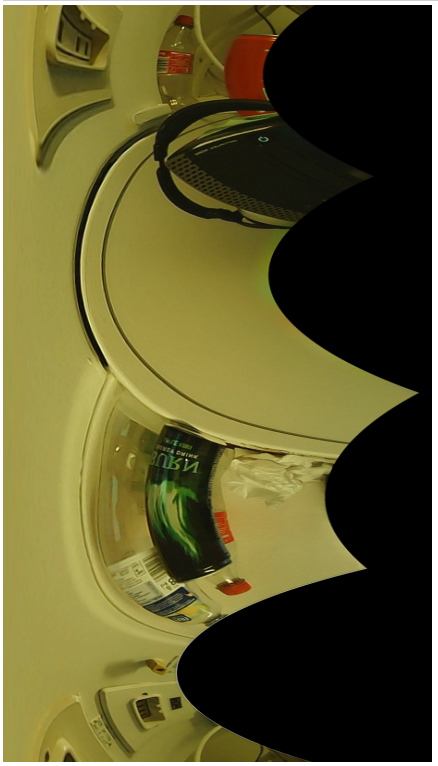
Figure 4.22 Two keyframes obtained by backward motion of the camera.

(a) Keyframe 1

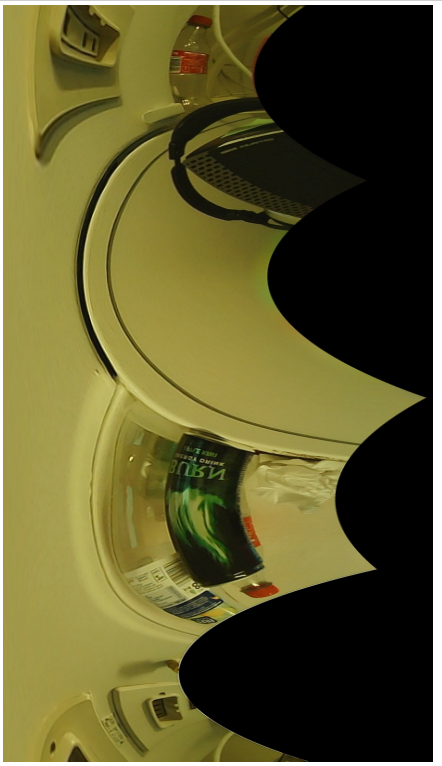


(b) Keyframe 2

Figure 4.23 The rectified keyframes obtained by backward motion of the camera



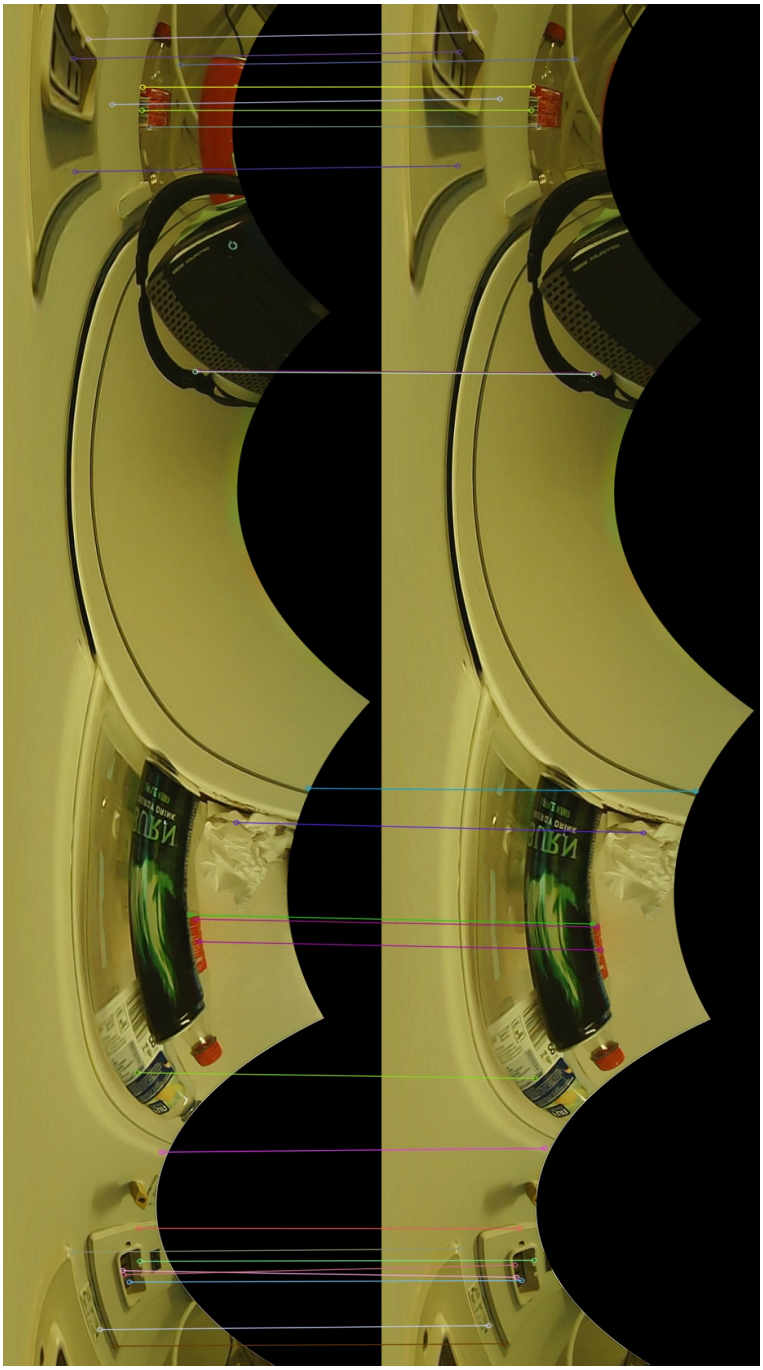
(a) Rectified Keyframe 1



(b) Rectified Keyframe 2

Figure 4.24 shows the corresponding points in the rectified image-pair.

Figure 4.24 Corresponding points between the rectified Keyframes



The differences in rows are shown in 4.25. For this particular experiment for backward motion the differences in rows were quite substantial. The minimum, maximum, average and median row differences are shown in table 4.6, and reveal a quite high average row difference.

Figure 4.25 The difference in rows between corresponding points in rectified image pair

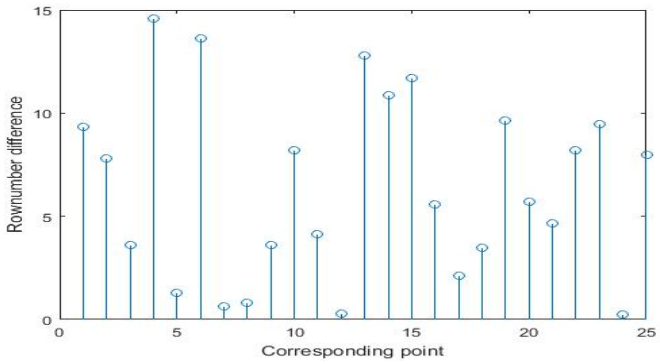


Table 4.6: Distance Statistics of Rectified Image pair

Difference in Rows Between Corresponding Matches (units are pixels)			
Min	**Max**	**Average**	**Median**
0.22	14.6	6.41	5.70

This experiment was done several times for both forward and backward motion. The computation time of the rectification was consistently between 15-22ms. Because the OpenCV matchers struggled with finding matches that were good, the image showing the corresponding points between the rectified keyframes required visual inspection to verify that the matches were correct. Thus, it was not feasible to let the algorithm run and compute the average of every image pair to reveal a trend. From the 20 or so experiments conducted, the average error in rows was usually ranging from 2-8 pixels, with the occasional very good result of below 1 pixel, and very bad result of above 10 pixels. Unfortunately, there was found no consistent trend of the row error being negative/positive for forward nor backward motion. The differences in rows for the rectified Keyframe 2 could both be above or below the ”corresponding” row in the rectified Keyframe 1. Finding a trend here could simplify the matching.

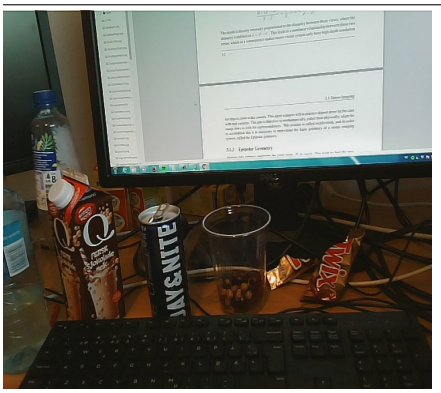
4.2.2 Left and Right Motion

Left Motion

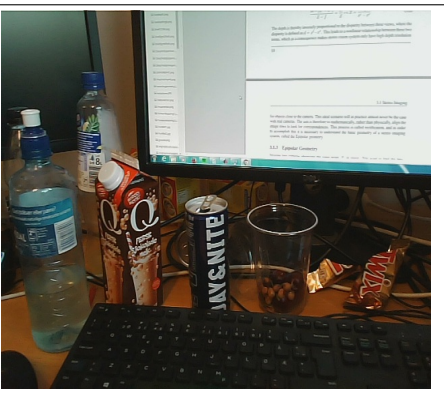
Figures 4.26 shows the Keyframes obtained by left motion of the camera. The rectified Keyframes are shown in figure 4.27. Because the epioles are located so far off the im-

age planes, the warping of the images are to a much lesser extent than with forward and backward motion. The rectification computational time was 8.662ms for this particular image-pair

Figure 4.26 Two keyframes obtained by left motion of the camera.

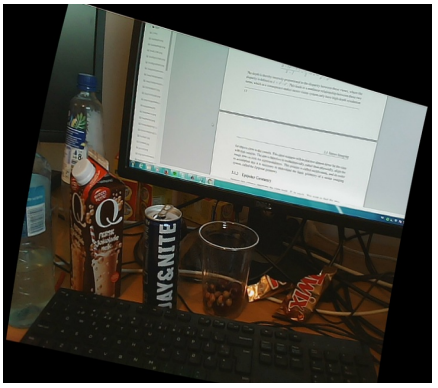


(a) Keyframe 1

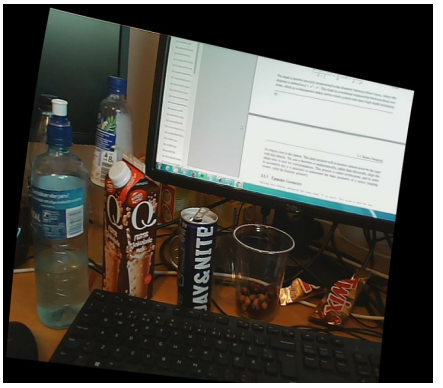


(b) Keyframe 2

Figure 4.27 The rectified keyframes obtained by left motion of the camera



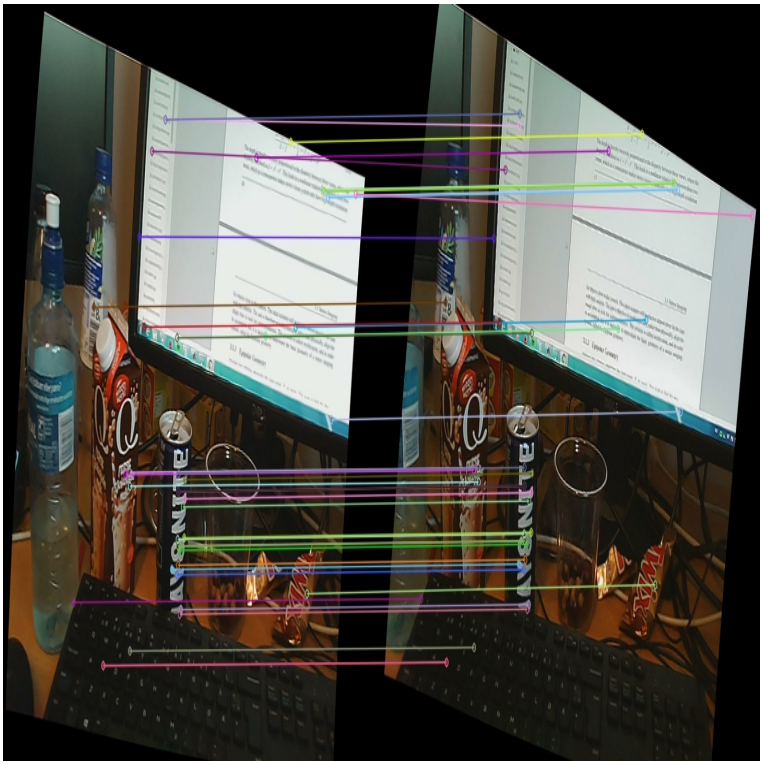
(a) Rectified Keyframe 1



(b) Rectified Keyframe 2

The corresponding points found by the OpenCV matcher are shown in figure 4.28.

Figure 4.28 Corresponding points between the rectified Keyframes



The plot in figure 4.29 shows the row errors for each of the point pairs found. The minimum, maximum, average and median errors are listed in table 4.7. While the average error is almost 5 rows difference, an interesting characteristic was discovered from this experiment. The row error is mostly positive from the rectified Keyframe 1 to the rectified Keyframe 2. Meaning, the corresponding row of any row in the rectified Keyframe 1 is most of the time located a few pixels above in the rectified Keyframe 2. This fact could be exploited to narrow down the search window in the stereo matching. In addition, the corresponding pixel of any pixel in Keyframe 1 is always located to the right in Keyframe 2, which also reduces the search window.

Figure 4.29 The difference in rows between corresponding points in rectified image pair

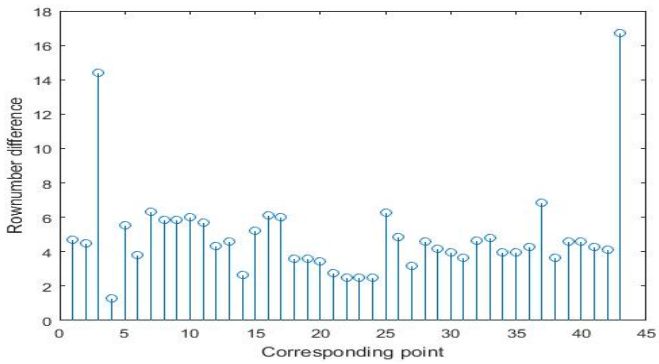


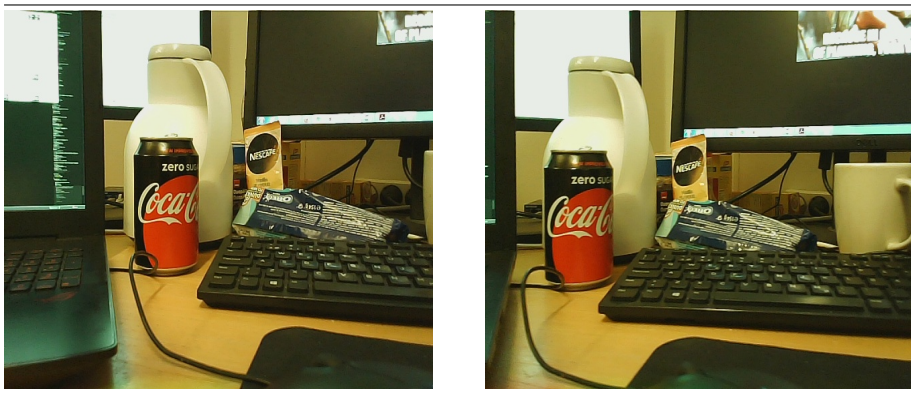
Table 4.7: Distance Statistics of Rectified Image pair

Difference in Rows Between Corresponding Matches (units are pixels)			
Min	**Max**	**Average**	**Median**
1.28	16.6	4.91	4.93

Right Motion

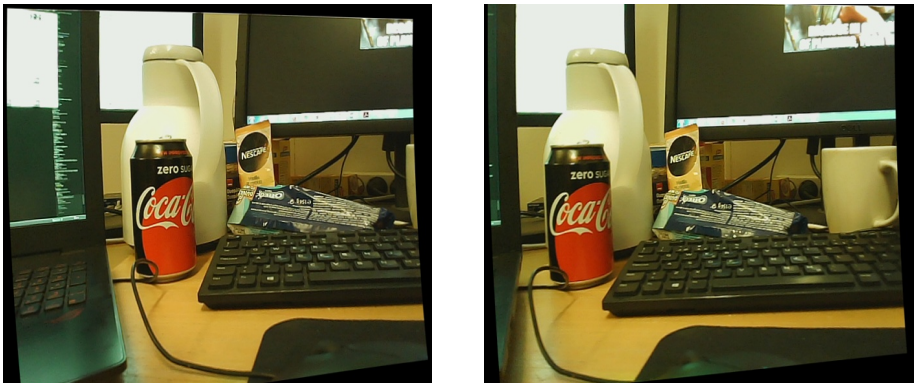
Figure 4.30 shows the Keyframes obtained by right motion of the camera, and their respective rectifications in figure 4.31. The computation of the rectified Keyframes took for this particular experiment 8.76ms.

Figure 4.30 Two keyframes obtained by right motion of the camera.



(a) Keyframe 1 **(b)** Keyframe 2

Figure 4.31 The rectified keyframes obtained by right motion of the camera

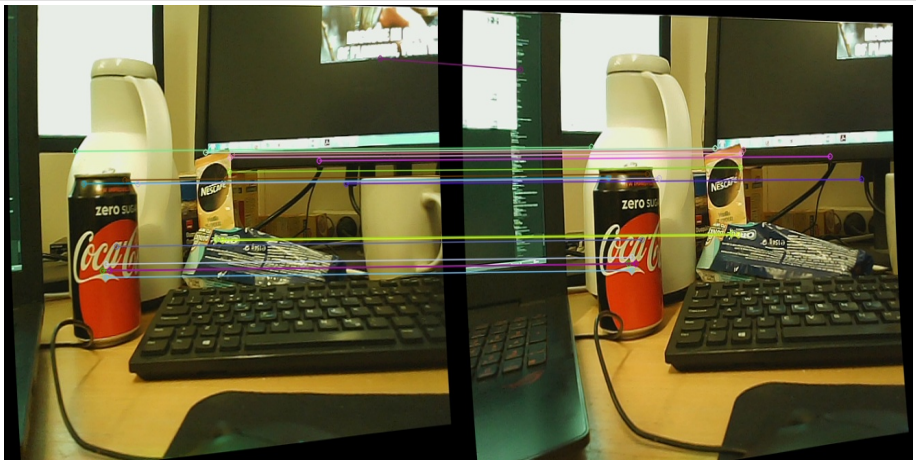


(a) Rectified Keyframe 1

(b) Rectified Keyframe 2

Figure 4.32 shows the point correspondences found by the OpenCV matcher.

Figure 4.32 Corresponding points between the rectified Keyframes



The plot in figure 4.33 shows the row errors for right motion, and the minimum, maximum, average and median errors are listed in table 4.8. For this particular instance, the errors are quite substantial. But similarly as with left motion, the experiment provided the discovery of the row error being positive from Keyframe 1 to Keyframe 2. And oppositely of left motion, any pixel in Keyframe 1 will lie to the left in Keyframe 2.

Figure 4.33 The difference in rows between corresponding points in rectified image pair

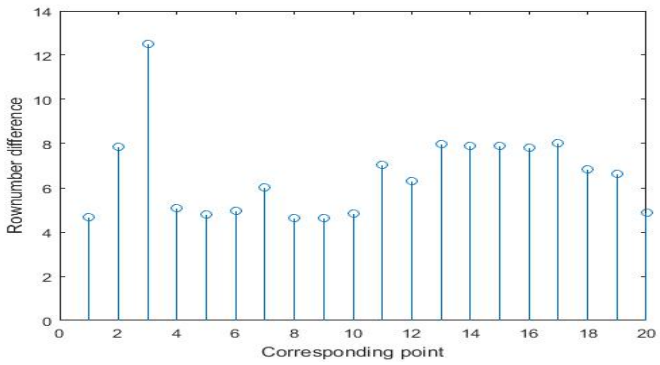


Table 4.8: Distance Statistics of Rectified Image pair

Difference in Rows Between Corresponding Matches (units are pixels)			
Min	**Max**	**Average**	**Median**
4.61	12.5	6.56	6.48

The experiment for left and right motion was repeated several times, as with forward/backward motion. The rectification computational time for left/right motion was found to be nearly identical, and always being somewhere around 8-10ms. The rectification of the Keyframes for left/right motion is thus consistently much faster than for forward/backward motion. This is not very surprising, as the rectified images for forward and backward motion are much larger in size. From the many experiments conducted, the average row error was found to be similar for left and right motion, ranging from 3-7 pixels, with the occasional better and worse instances.

4.2.3 Assessment of the Rectification

The errors from the Fundamental Matrix was found to propagate, as expected, into the rectification. This will have consequences for the stereo matching, as the corresponding image points can not be expected to lie on the same rows. The reasons for why the errors are present in the rectification is similar to the errors in the Fundamental Matrix.

- Inaccurate tracking information from Kintinuous.
- Errors in the matches found by the OpenCV descriptor matcher.
- Shaking of the authors hand while holding the camera, resulting in images not being at the exact position deduced by Kintinuous.
- Imperfections in the implementation of the rectification algorithm is also a possibility.

In contrast to the accuracy of the Fundamental Matrix, distortion is not a factor affecting the error in the rectification, as the images are undistorted before being rectified. Errors from the Kintinuous tracking information is probably the biggest cause for errors, as with the Fundamental Matrix.

4.3 Stereo Matching

Due to the inaccuracy of the rectification, the disparity map computation proved itself extremely difficult. Dozens of different approaches were tried, taking processing time and the resulting disparity map into account when assessing every approach. What stayed consistent for every approach was the choice of descriptor type when performing the matching. Only fast descriptor types were utilized, such as ORB, BRIEF or DAISY, to reduce the computation time. For the same reason, no global matching methods were explored.

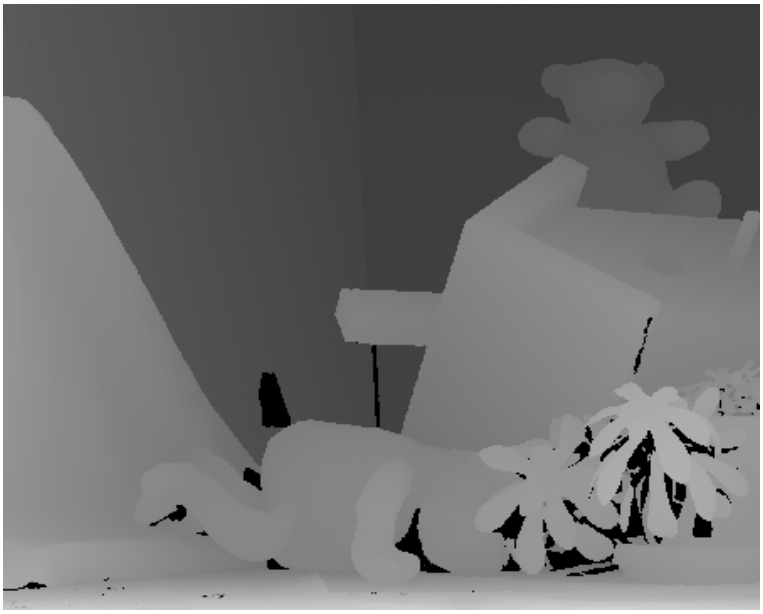
Because the ground truth disparity of the disparity maps computed from the Kintinuous images are unknown, a perfectly rectified image-pair from the Middlebury dataset was used to assess the different approaches. The image pair used from the Middlebury dataset is shown in figure 4.34, and its ground truth disparity map in figure 4.35.

Figure 4.34 The perfectly rectified stereo image pair from the Middelbury dataset used in the development of the stereo matching algorithm.

(a) Left image

(b) right image

Figure 4.35 The ground truth disparity map of the Teddy stereo image pair.



The rectified image-pair from Kintinuous used for testing is shown in figure 4.36. These images were chosen because the rectification was not very accurate, where corresponding points could be found up to 6 rows above or below the current row. In addition, one of the images is quite noisy, which might often be the case when the image is captured while the camera is moving. If the matching algorithm performs well under these conditions, it should at least perform just as well under better conditions.

Figure 4.36 The rectified Keyframes used for testing of the matching algorithms.

(a) Left rectified Keyframe used for testing

(b) right rectified Keyframe used for testing

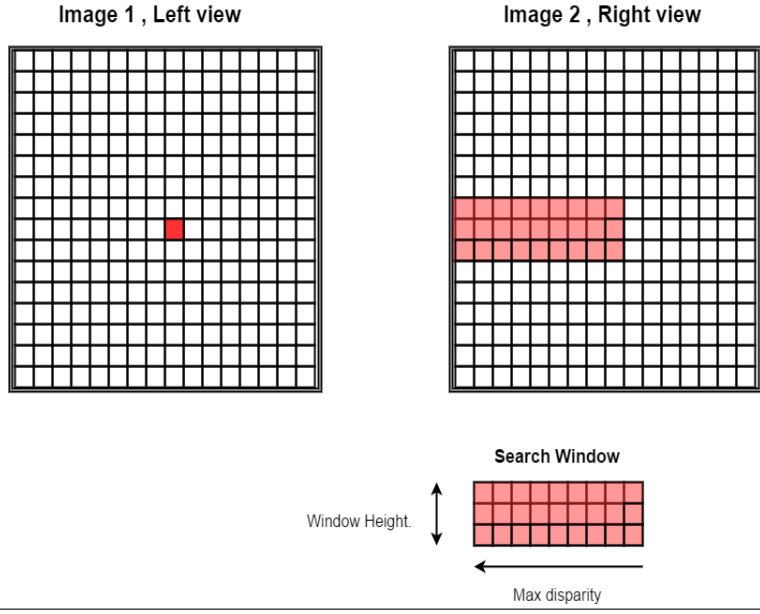
Sections 4.1-4.3 will explain and review a select few of the approaches that were tried

and discarded. The reason for providing the results of discarded matching approaches is to give insight into why the final matching algorithm ended up as it did. Section 4.5 will provide the result of the actual matching algorithm implemented in the system.

4.3.1 Window Matching With ORB/BRIEF and Crosscheck

One of the first approaches conducted was window search with ORB and BRIEF. Every pixel in Image 1 is compared to every pixel in Image 2 that lies inside some window. Several window sizes were experimented with, with regards to the height of the windows, because corresponding points in Image 2 can lie several rows above and below the current row in Image 1. The width of the window must be the maximum possible disparity, which was found to be 170, as shown in section 3.4.3. The match for each search was found by calculating the Hamming distance between the descriptor of each pixel pair, where the best match would have the smallest distance. The search window is illustrated in figure 4.37

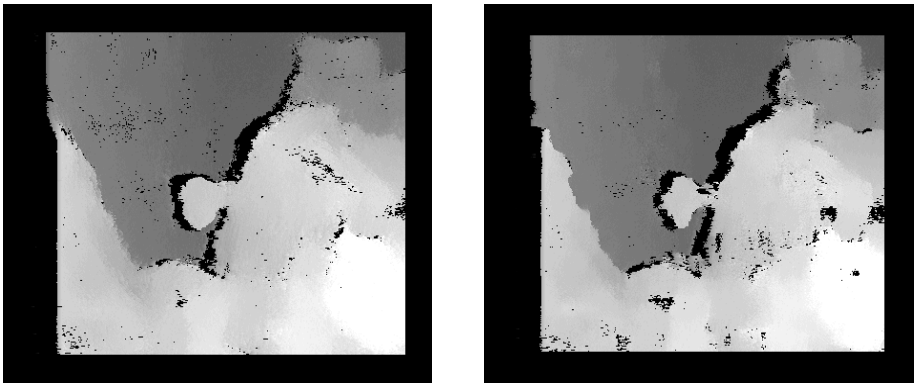
Figure 4.37 The red pixel in image 1 is being processed. The right rectangle in Image 2 is the search window. The width of the search window is the max number of disparities. The height of the search window is the number of rows being searched above and below the current row. The best match is found by finding the descriptor pair closest in Hamming distance.



A bigger height search window naturally led to a much longer computation time. Searching 5 rows above and below the current row led to the computational time being several minutes. After several tests with different search window heights, it showed no major difference in getting a more accurate disparity map, only longer computational time.

The decision was therefore made to stick to the same row in both images, with the search window then being of size [1][Max disparity]. Due to occlusions and the non-overlapping regions in the images, a crosscheck was implemented to remove bad matches. For the current pixel, $pixel_{current}$, in the left image, a match is found in the right image, denoted $match_{right}$. Then, the process is reverted, finding the pixel in the left image that best matches $match_{right}$. If the left match found is located in close proximity to $pixel_{current}$, the match is accepted and the disparity value calculated. If not, the match is rejected. The resulting disparity map of the Teddy images for this approach is shown in figure 4.38. The computation time was about 3.75 seconds for both ORB and BRIEF. The resulting disparity map for the rectified Kintinuous images are shown in figure **??**. The computation time was 33.1 seconds for BRIEF, and 34.0 seconds for ORB. Since the maximum disparity in the Teddy images is only about 40 disparities, and the images are much smaller in size than the Kintinuous images, the computation time is naturally much lower.

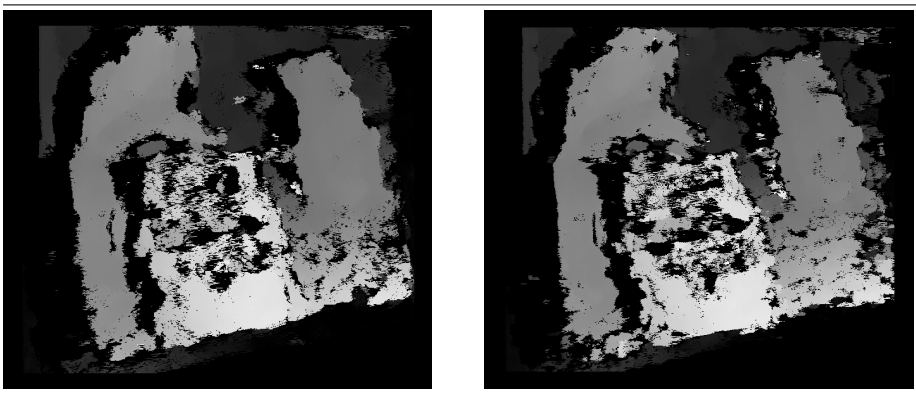
Figure 4.38 Disparity map using Brief/ORB, windows search and crosscheck



(a) Brief

(b) Orb

Figure 4.39 Disparity map of Kintinuous Keyframes using Brief/ORB, windows search and crosscheck



(a) Brief **(b)** Orb

The similarity score of BRIEF and ORB to the ground truth of the Teddy images was 0.653177 and 0.653137, respectively. Due to BRIEFs slight edge in terms of speed and smoother disparities, it was mostly used instead of ORB when testing the different matching algorithms. This approach was discarded due to the high degree of noise in the disparity images and the long computational time.

4.3.2 Variable Window Size

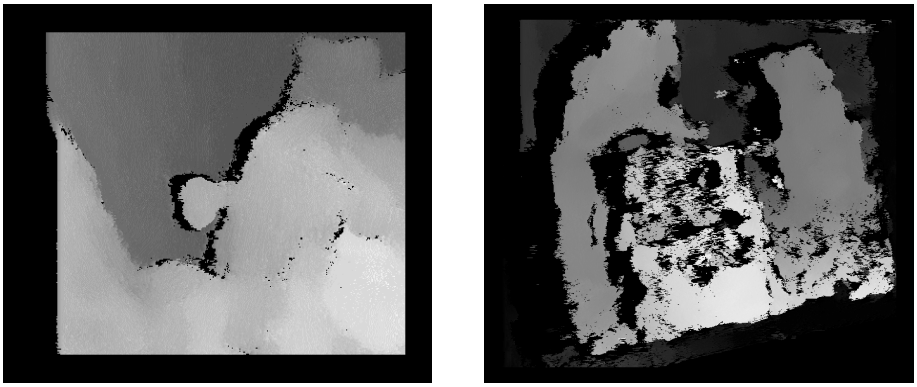
This approach tried to reduce the computational time by introducing a variable search window. By assuming that the disparity does not change much from one pixel to the next, the search can be restricted to a small window around the previous match found.

```
1 For all rows in left image
2  For all columns in left image
3    If (previous match = NULL)
4      Find the match of pixel [row][col] in full search window
5      Crosscheck match
6      If (Crosscheck is passed)
7         previous match = match
8         set disparity  = abs(col - match)
9      Else
10        set disparity = 0
11        previous match = NULL
12   Else
13     Find the 2 best matches of pixel [row][col] in a small
                  search window around the previous match
14     Perform uniqueness ratio test on match
15     If (uniqueness test is passed)
16       previous match = match
```

```
17          set disparity = abs(col - match)
18        Else
19          jump to 4
```

The uniqueness test mentioned in the pseudocode above is to filter out a bad match when finding it in the small window. The small window is only 10 pixels wide, so performing a crosscheck would not guarantee a safe filtering. The uniqueness test detects potential bad matches by comparing the cost of the best match to the cost of the second best match. If their costs are close to equal, the match is rejected. If the cost of the best match is much smaller than the cost of the second best match, the match is accepted due to its uniqueness. The disparity maps produced by this approach are shown in figure 4.40. For the Kintinuous images, this approach spent 24 seconds computing the disparity map, thereby beating the previous approach by almost 10 seconds. Of the total 457 646 pixels in the rectified Keyframe, 14835 disparity values were set by searching in the small window. The similarity score of the Teddy disparity map with regards to the ground truth was 0.720491.

Figure 4.40 Disparity map of Kintinuous Keyframes using Brief with variable window search.

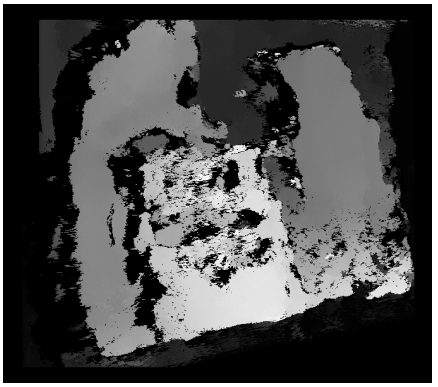


(a) Teddy disparity map

(b) Kintinuous Keyframe disparity map

This approach was also tested with searching several rows above and below the current row to see if it improved the disparity map. Searching 5 rows above and below each current row increased the computation time to over 80 seconds, where 10516 disparity values were set in the small window. Searching 10 rows above and below the current row increased the computation time to 162 seconds, where 10958 disparity values were set in the small window. The resulting disparity maps can be seen in figure 4.41. Arguably, the disparity maps improve very slighty when searching above and below the current row. However, the drastic increase in computation time makes it unaccaptable.

Figure 4.41 Disparity map of Kintinuous Keyframes using Brief with variable window search, with several rows window height.



(a) 5 rows above and below search

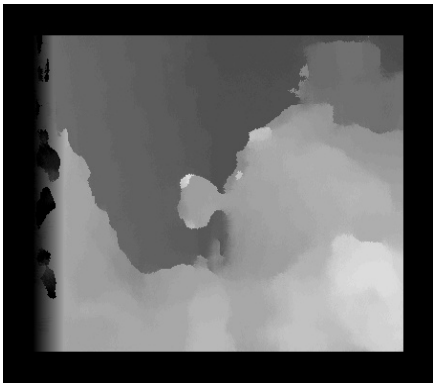
(b) 10 rows above and below search

4.3.3 Semi-Global Matching

A simple version of Semi-Global matching(CITE) implemented by github user ”reisub” (CITE) was tested and extended to allow for using descriptors to calculate the pixel costs. The cost volumes created by this implementation contain one disparity candidate for all pixels in the max disparity range. For the Teddy images, this means that the initial cost volume will be of size [height][width][40], and for the Kintinuous rectified Keyframes, [height][width][170]. The cost is aggregated so that every disparity candidate influences the final cost volume. Naturally, this takes a long time. Figure **??** shows the disparity maps of the Teddy image-pair while using individual pixel intensities as costs, and while using the pixels ORB descriptors as costs. The computation of the Teddy disparity map while using individual pixel intensities took 15.68 seconds, and had a ground truth similarity score of 85%. The Teddy disparity map while using ORB descriptors took 22.87 seconds and had a ground truth similarity score of 76%. The disparity maps are shown in figure 4.42. As seen in the figure, using individual pixel intensities as costs result in much sharper edges. This is due to the fact that descriptors describe each pixel by looking in its neighborhood. What might happen, and is very visible on the chimney of the house lying in the middle of the picture, is that the neighborhood causes faulty matches. The blue painting located in the background affects the descriptors of the pixels on the chimney. The correct correspondences of the chimney might not ”look” like each other in terms descriptors because of the blue paintings contribution. This is one of the factors why the edges look ”soft” when using the descriptors. Also, remember that ORB and BRIEF can not be calculated close to the edges of the images. Therefore, all disparity values close to the edges will be zeroed out. This affects the similarity score to some degree.

Figure 4.42 Disparity map of Teddy-image pair from Semi-Global matching

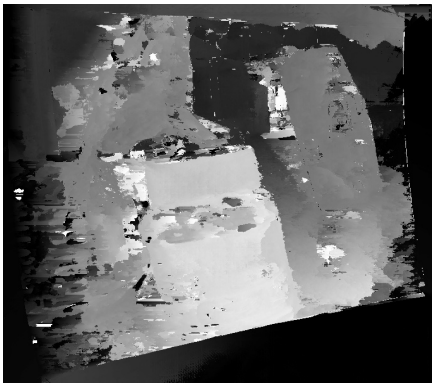
(a) Individual pixel intensity cost



(b) Descriptor cost

The disparity maps for the Kintinuous images computed by the simple SGM for both individual pixel intensities and descriptors are shown in figure 4.43. The computation of the disparity map with individual pixel intensities as costs took 325 seconds, and for descriptors 569 seconds. Needless to say, the computation time is quite substantial. Similarly to the Teddy disparity maps, using individual intensities as costs make some edges sharper for the Kintinuous images as well. However, not every edge is captured, and the resulting disparity map is very noisy. The same can be said for the disparity map computed by comparing descriptors.

Figure 4.43 Disparity map of rectified Kintinuous Keyframes from Semi-Global matching



(a) Individual pixel intensity cost

(b) Descriptor cost

4.3.4 Matching System Component Results

The assessment of the final matching algorithm was conducted both offline, using the Teddy and Kintinuous testing images, and online while Kintinuous was running. The

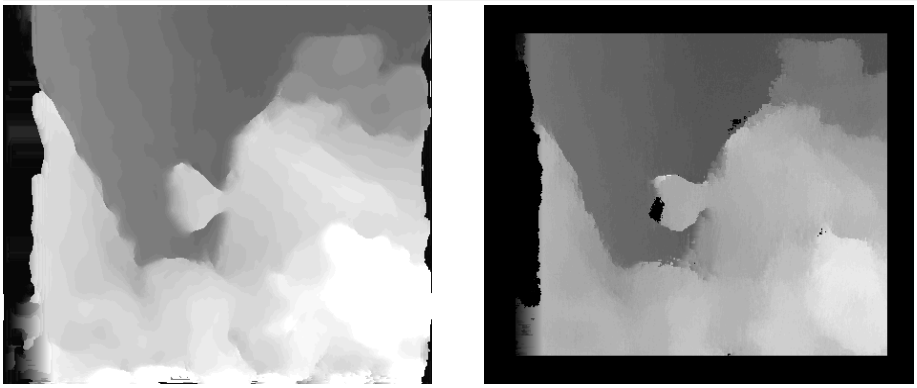
offline experiments will be presented first, followed by the online experiments.

Offline Experiments

The final matching algorithm was first assessed in the same fashion as the previous matching methods. It was tested using ORB, BRIEF and DAISY on both the Teddy and Kintinuous image-pairs.

The disparity map of the Teddy images, using DAISY and Brief, is shown in figure 4.44. The computation time using DAISY was 9.28 seconds, and the ground truth similarity score only 67%. The computation time using BRIEF was 2.99 seconds, beating all the previous approaches in terms of computational time, with a similarity score of 73%. The computation of the DAISY descriptors is quite time consuming in contrast to BRIEF, and is the reason why the matching takes so much longer.

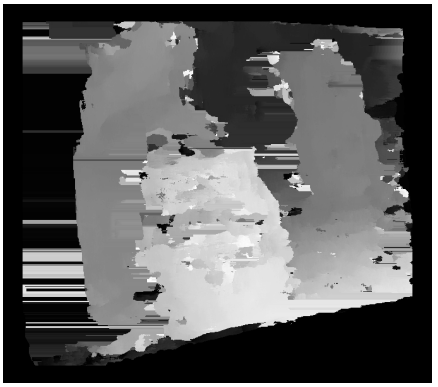
Figure 4.44 Disparity map of Teddy images from final matching algorithm



(a) DAISY descriptor as cost **(b)** BRIEF descriptor as cost

The computation time while using ORB and BRIEF descriptors as the costs resulted in a computation time of approximately 13 seconds on the Kintinuous testing images. The computation time while using the DAISY descriptor was 21.3 seconds. Figure 4.45 shows the disparity maps when using ORB and BRIEF, and figure 4.46 when using DAISY. Table 4.9 shows the computation time for the various parts of the algorithm. The reason why this algorithm is so much faster than the previous approaches in the sections above is because of the OpenCV bruteforce matchers matching every pixel on both rows at the same time. Even though this algorithm has a quite time consuming cost aggregation step, the total computation time when using ORB and BRIEF is half of what the previous approaches spent. The reason for choosing DAISY over ORB and BRIEF, even though it is much slower, is the quality of the computed disparity maps from the Kintinuous Keyframes. The disparity is by no means perfect, and still contain a high degree of noise, but the edges are sharper, and the disparities more continuous.

Figure 4.45 Disparity map of Kintinuous images from final matching algorithm



(a) ORB descriptor as cost

(b) BRIEF descriptor as cost

Figure 4.46 Disparity map of Kintinuous images from final matching algorithm using DAISY descriptors as cost

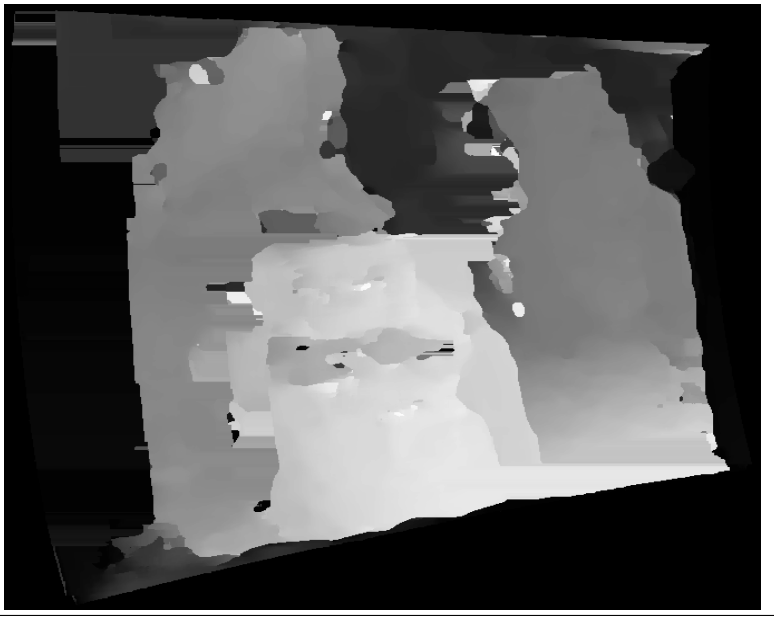


Figure 4.47 shows the disparity map when the cost aggregation is not performed.

Figure 4.47 Disparity map without cost aggregation

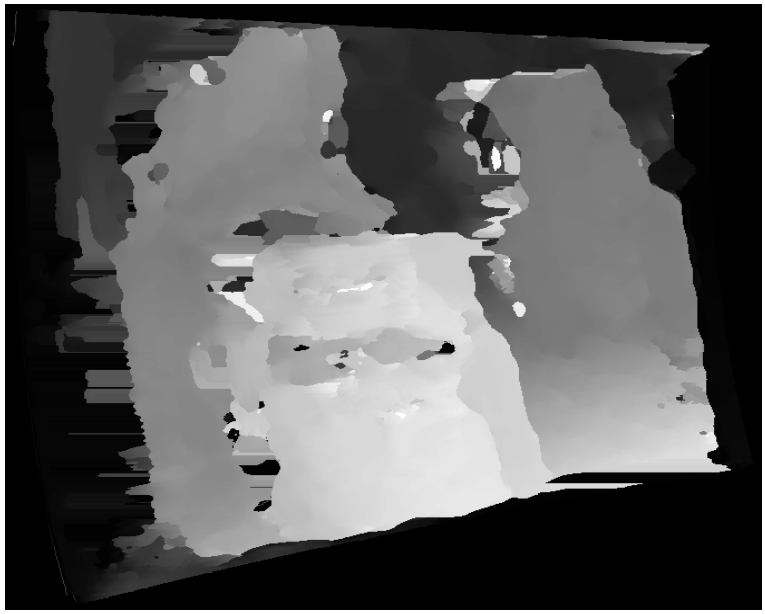


Table 4.9: Computational times of matching algorithm components

The individual components computation times (unit is seconds)				
Descriptor Type	**Descriptor Computation**	**Cost Volume**	**Cost Aggregation**	**Total**
ORB	2.79	2.36	7.44	13.06
BRIEF	1.9	2.89	7.53	12.77
DAISY	9.21	3.29	7.79	21.29

Online Experiments

The online experiments were conducted similarly to when assessing the Fundamental Matrix and the rectification. The camera was held by hand, emulating a snake robots head while moving it. The interesting scenarios that were looked for was again how the different locations of the epipoles affected the matching process. Unfortunately, none of the matching algorithms tried performed consistently well when implemented into Kintinuous. Because of this, the results in this section is a bit lackluster, as they were done on the very end of the thesis, in hope that it would work at last. Figure 4.48 shows the Keyframes obtained by right motion of the camera, while figure 4.49 shows the rectified Keyframes.

Figure 4.48 Keyframes from right motion of the camera

(a) Left view

(b) Right view

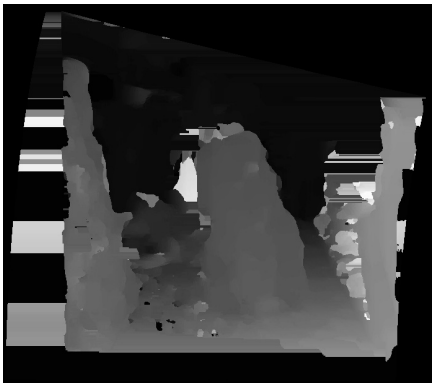
Figure 4.49 Rectified keyframes from right motion of the camera

(a) Left view

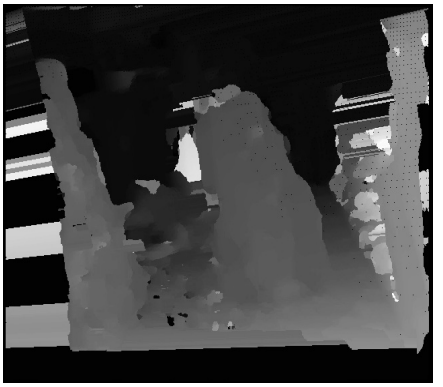
(b) Right view

The disparity map was computed by using the final matching algorithm, and then remapped with the maps from the rectification component to obtain the unrectified disparitymap. These can be seen in figure 4.50. The disparity maps looks quite fine in the middle, but there is a high degree of noise and streaking going on around the edges. This was the best the disparity maps got.

Figure 4.50 Disparity maps from right motion of the camera



(a) Rectified disparity map



(b) Unrectified disparity map

For forward and backward motion, the disparity maps were completely useless due to the bad continuity and noise. Such a case can be seen in the following figures 4.51, 4.52 and 4.53.

Figure 4.51 Keyframes from forward motion of the camera

(a) First view

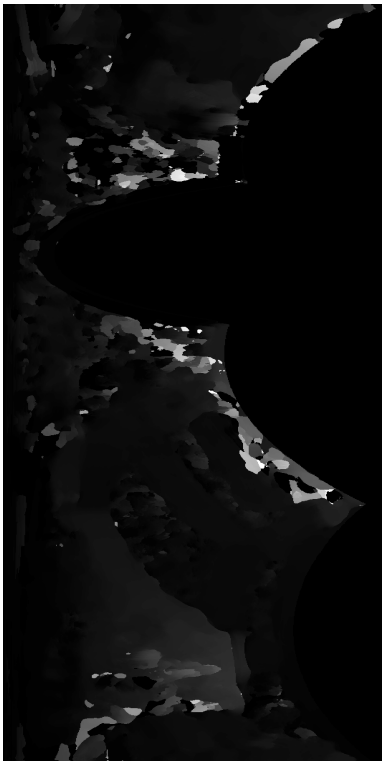
(b) Second view

Figure 4.52 Rectified keyframes from forward motion of the camera

(a) First view rectified

(b) Second view rectified

Figure 4.53 Disparity map from forward motion



One final example is shown in figures 4.54, 4.55 4.56, this obtained from left motion.

Figure 4.54 Keyframes from left motion of the camera

(a) Right view

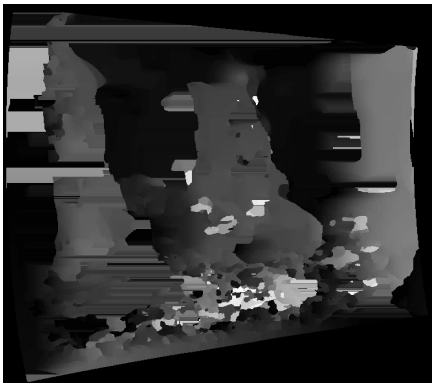
(b) Left view

Figure 4.55 Rectified keyframes from left motion of the camera

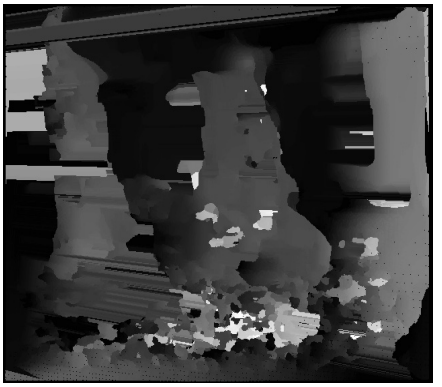
(a) Right view

(b) Left view

Figure 4.56 Disparity maps from left motion of the camera



(a) Rectified disparity map



(b) Unrectified disparity map

CHAPTER 5

Discussion

This chapter will serve as more of a general discussion on the approach taken when working on this thesis, and the results in the previous chapter.

5.1 Goal and Method

The goal of this thesis was to compensate for the Kinects limited range by using the single RGB-camera on the sensor, exploiting Kintinuous tracking information, and use what is applicable from stereo vision. The goal as a whole was divided into several smaller objectives that had to be done successively; the Keyframe acquisition, Fundamental Matrix computation, rectification, stereo matching and depth map fusing. Except for the depth map fusing, every objective was dependent on having a solid understanding of stereo vision theory to know what difficulties to expect, how to tackle these difficulties, and to be prepared to handle unexpected problems. Having a solid grasp of the Kintinuous source code and data flow was imperative for every single objective. The depth map fusing was solely dependent on understanding Kintinuous. All objectives were tackled with the same approach; start with a simple concept that compiles and runs, and add to this concept as time progress until at solution presents itself. If problems occur, perform experiments to locate the source of the problem and go on from there.

The Keyframe acquisition, Fundamental Matrix computation and rectification objectives were all handled in a straight forward manner. First, a thorough understanding of the stereo vision theory was acquired, and the necessary parts of the Kintinuous source code needed for the actual implementation studied. Then, Kintinuous was expanded with a new system component with access to the necessary data structures. The realizations of the first three objectives were performed in successive order, implementing them directly into the new Kintinuous system component, as they were directly dependent on the information Kintinuous provided online. The implementation of these three first objectives

presented no difficulties worthy of mention. To the naked eye, it seemed to be working really well. However, performing in depth experiments to assess the performance of these objective were not done until the struggles of the matching process became overwhelming. This was a big mistake because not having an in depth overview of the imperfections of the components that the matching process depended on, made it difficult to know why it failed. If these experiments were performed earlier, some of the struggles experienced in the early development of the matching algorithm could probably be avoided. This is an important lesson.

The development of the matching algorithm was done outside of Kintinuous by storing rectified Keyframes and experimenting with different matching approaches in an offline testing environment. This consisted of getting familiar with OpenCV, and writing simple matching algorithms computing the disparity maps of a perfectly rectified image pair, before moving on to the Kintinuous images. Developing a matching algorithm that was able to handle the inaccurate rectifications proved itself to by extremely difficult, and thereby very time consuming. After dozens of different approaches, totaling over 10000 lines of code, the time restriction of the thesis forced the choosing of the final approach. Even though the final matching algorithm performed decently enough on the testing images, its performance when used online in Kintinuous struggled greatly. This led to the actual fusing part not getting tested, and consequently the main goal of the thesis not being accomplished.

In hindsight it may have been unwise to try to develop a matching algorithm from scratch for such a long period of time, before turning to existing well studied methods. There are matching methods designed to deal with noise, occlusions and even unrectified stereo images. Exploring these methods as this systems matching algorithm might have led to a functioning implementation. A big part of practical projects like this is time spent on the error searching, trying and failing. As a side note, spending lots of time solving errors and problems can be very educational; as long as they are resolved, that is.

Nonetheless, the goal of this thesis is very interesting. The work done and findings presented here form a basis for further work.

5.2 Result

The goal was to compensate for the Kinects limited range by obtaining dense depth maps by stereo vision principles, and fusing them with the environment generated by Kintinuous. Unfortunately, the difficulties experienced with the development of a matching algorithm resulted in time running out before the depth map fusing could be tested and evaluated. However, the execution of each individual component composing the system, up until the matching, performed as intended.
The experiments used to assess the computation of the Fundamental Matrix showed the Kintinuous tracking information to provide better geometrical relations between the views than the OpenCV 8-point algorithm. At least when only using 8 point correspondences to compute the matrix. It is quite possible that the 8-point method would perform better than Kintinuous if it was given more point correspondences to constrain the calculation, especially if outliers were to be eliminated by for example RANSAC. If the 8-point method was to be used for the Fundamental Matrix computation in the system, however, process-

ing time would increase as point correspondences would have to be computed and filtered. Also, good point correspondences are not always guaranteed when capturing low-texture areas, making the Kintinuous Fundamental Matrix the safer choice. The accuracy of the Fundamental Matrix was shown to be very variable. In some cases, the distances from the points to the epipolar lines could be close to zero pixels, and in other cases almost 10 pixels. There seemed to be no consistent indication of the Fundamental Matrix accuracy being any worse or better for any of the directions tested. The reason for the bad accuracy is most likely that the orientation data from Kintinuous is not perfect. If the accuracy of the Fundamental Matrix proved itself to be extremely good and to the point were the stereo matching no longer is a problem, by using the 8-point method with more point correspondences, this is worth being explored. By choosing this approach, the number of confidence point correspondences used in the computation of the Fundamental Matrix would have to be counted. Depending on how many good matches there were, a decision could be made about if the Fundamental Matrix should be trusted or not. If not, discard the matrix and obtain two new views. Another possible solution could be to have an outside sensor keeping track of the head of the snake robot, using for example object tracking, telling Kintinuous when to grab a new Keyframe. This radical approach, if working, could solve many problems in the system. However, it would completely contradict the independence of the snake robot.

The rectification was shown to be very fast, with a slightly longer computational time when performing the rectification during forward/backward motion compared to left/right motion. As expected, the inaccuracy of the Fundamental Matrix carried over into the rectification. A more thorough evaluation of the rectification could be performed to see if the inaccuracy of the rectification is the exact same as the inaccuracy of the Fundamental Matrix used in the computation. This would show if the actual rectification algorithm worsens the accuracy, or if the error is purely propagated from the Fundamental Matrix. In the case of the former, the implementation of the rectification algorithm should be reviewed, or even try to implement a different rectification algorithm. ”A compact algorithm for rectification of stereo pairs” [24] is a simple algorithm designed for dealing with the rectification of an unconstrained stereo rig. This could be a viable alternative.

The development of a capable stereo matching algorithm was sadly not overcome. The algorithm as it stands, does not handle occlusions, produces noisy disparity maps in most cases, and completely fails in computing disparity maps for forward and backward motion. Running the matching algorithm on the testing images showed decent results, but they did not carry over to consistent behaviour when implemented to Kintinuous. With the inaccuracy of the rectification resulting on conjugate points being several pixels above or below each other, the hope was to be able to compensate for this by searching above and below each line for every match. However, this led to a very costly computational time and bad matches increased. Different kinds of descriptors were tested with the hope that each pixel’s neighborhood coherently being included in the descriptors should make the matching work better. Descriptors performed better, but a satisfactory matching algorithm was never achieved. This in turn led to the depth fusing never being tested.

Even though the depth fusing was never tested, it is still possible to discuss. The two approaches proposed was to either feed the depth map into Kintinuous in the same manner as the Kintinuous depth maps are received, or to create a point cloud using the disparity map,

and fuse this with the point cloud being created by Kintinuous. The latter would probably be more tricky to perform, while the former very easy, as long as Kintinuous allows the use of this pipeline to carry the stereo depth maps. What might happen when inserting the depth map, or point cloud, into Kintinuous is that it looses track due to the sudden change in the point cloud. Another possibility is that it works well, the new 3D surface gets fused with the existing one, but then fades over time. If objects that Kintinuous is tracking are removed from the scene, they will slowly but surely be filtered away.
During the execution of the experiments there was discovered a problem that might affect the decision of using Kintinuous as the snake robot's SLAM system. The goal is to compensate for the limited range by creating depth maps while exploiting tracking information. This in itself is in the worst cases a chicken and egg problem. When the Kinect sensor is operating too close to produce high resolution depth maps, but still is far away so that decent tracking is achieved, the resulting Fundamental Matrix produces a decent rectification. However, when the Kinect sensor is too close, the tracking is thrown completely off track, resulting in very bad rectification which could not possibly result in a decent depth map. This is something that should be considered before eventually continuing the work presented here.

5.3 Future Work

The system requires a well functioning matching algorithm for the fusing of the depth maps to be performed. Research should be done in finding a well studied and fast matching algorithm that could potentially work with the bad rectifications. Or, implement a matching algorithm that is meant to find correspondences in unrectified images. If the matching algorithm performs well enough, the depth maps should be fused with Kintinuous, either by one of the approaches suggested, or in some other way.

CHAPTER 6

Conclusion

This thesis has investigated the possibility of compensating for the Kinect sensor's limited range by the use of stereo vision principles, the single RGB-camera on the Kinect, and the tracking information provided by Kintinuous. A new system component was implemented to Kintinuous responsible for every part of the stereo depth map creation. This consist of saving two different view images from the Kinect seperated by a small baseline, perform the rectification using the tracking information provided by Kintinuous, and match every pixel in the rectified images to produce a disparity map. The range compensation was not successful due to the bad quality of the resulting depth map. Experiments conducted showed that the tracking information provided by Kintinuous led to an inaccurate rectificaton of the images which in turn caused many problems for computation of a disparity map. The matching algorithm responsible for the computation of the disparity map was unable to consistently match the correct pixels, leading to noisy or unrecognizable disparity measurements.

Bibliography

[1] OpenCV Documentation. Understanding features.

[2] Gary Bradski and Adrian Kaehler. *Learning OpenCV*. 2008.

[3] R. Koch M. Pollefeys and L. Van Gool. A simple and efficient rectification method for general motion. Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference, 1999.

[4] Middlebury. Middlebury stereo datasets.

[5] Michael Kaess Maurice Fallon Hordur Johannsson John J. L eonard Thomas Whelan, John McDonald. Kintinuous: Spatially extended kinectfusion. 2012.

[6] K. Y. Pettersen A. A. Transeth and P. Liljebäck. A survey on snake robot modeling and locomotion. In *Robotica, vol. 27, no. 07*, 2009.

[7] Filippo Sanfilippo, Øyvind Stavdahl, Giancarlo Marafioti, Aksel A. Transeth, and Pål Liljebäck. Virtual functional segmentation of snake robots for perception-driven obstacle-aided locomotion. In *Proc. of the IEEE Conference on Robotics and Biomimetics (ROBIO), Qingdao, China*, 2016.

[8] Filippo Sanfilippo, Jon Azpiazu, Giancarlo Marafioti, Aksel A. Transeth, Øyvind Stavdahl, and Pål Liljebäck. A review on perception-driven obstacle-aided locomotion for snake robots. In *Proc. of the 14th International Conference on Control, Automation, Robotics and Vision (ICARCV), Phuket, Thailand*, 2016.

[9] Fredrik Leine Lerø. Environment representation for a snake robot. In *NTNU*, 2016.

[10] Otmar Hilliges David Molyneaux David Kim Andrew J. Davison Pushmeet Kohli Jamie Shotton Steve Hodges Andrew Fitzgibbon Richard A. Newcombe, Shahram Izadi. Kinectfusion: Real-time dense surface mapping and tracking. 2011.

[11] Savide Scaramuzza and Friedrich Fraundorfer. Visual odometry part i: The first 30 years and fundamentals. In *IEEE Robotics and Automation Magazine, Volume 18, issue 4*, 2011.

[12] Computer Vision Group. Visual slam.

[13] J. M. M. Montiel Raúl Mur-Artal and Juan D. Tardós. Orb-slam: A versatile and accurate monocular slam system. In *IEEE Transactions on Robotics, vol. 31, no. 5, pp. 1147-1163, October*, 2015.

[14] D. Cremers J. Engel, T. Schöps. Lsd-slam: Large-scale direct monocular slam. In *European Conference on Computer Vision (ECCV)*, 2014.

[15] Pauline Tan Vladimir Kolmogorov, Pascal Monasse. Kolmogorov and zabih's graph cuts stereo matching algorithm. In *Image Processing on Line, 4, pp.220-251*, 2014.

[16] Richard E. Woods Rafael C. Gonzalez. *Digital Image Processing, third edition*. Pearson, 2008.

[17] OpenCV Documentation. Feature detection and description.

[18] Marco Trebeschi Giovanna Sansoni and Franco Docchio. State-of-the-art and applications of 3d imaging sensors in industry, cultural heritage, medicine, and criminal investigation.

[19] Orbbec. Orbbec astra pro.

[20] NVIDIA. Cuda.

[21] D. Galvez-Lopez and J. D. Tardos. Real-time loop detection with bags of binary words. In *In Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference*, 2011.

[22] H. Hirschmüller. Stereo processing by semi-global matching and mutual information. In *vol. 30, no. 2, pp. 328-341*, 2008.

[23] Pascal Fua Engin Tola, Vincent Lepetit. Daisy: An efficient dense descriptor applied to wide baseline stereo. In *IEEE Transactions on Pattern Analysis and Machine Intelligence Vol. 32, Nr. 5, pp. 815 - 830, May 2010*, 2010.

[24] Trucco E. Verri A. Fusiello, A. A compact algorithm for rectification of stereo pairs. In *Machine Vision and Applications*, 1999.

Appendix

A digital attachment includes the source code of the extended Kintinuous algorithm. The contribution is found in the class ”StereoDepth”, under src/utils.