Creating 360 View by Combining Multiple Fish-eye Images

- Master's Thesis -

Project Report

Freddy Gramstrup Nielsen

Aalborg University
Vision, Graphics, and Interactive Systems
2016

Copyright © Aalborg University 2016



Electronics and IT
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Creating 360 View by Combining Multiple Fish-eye Images

Theme:
Master's Thesis

Project Period:
Spring Semester 2016

Project Group:
16gr1045

Participant(s):
Freddy Gramstrup Nielsen

Supervisor(s):
Prof. Thomas B. Moeslund
Aalborg University
Dr. Salvatore Livatino
University of Hertfordshire

Copies: 1

Page Numbers: 64

Date of Completion:
June 2, 2016

Abstract:

Compositing images from multiple cameras mounted on a car to create a bird's-eye view to assist a driver while parking is a relatively new idea. Though it has been researched in the past, it is only in recent years that car makers have begun selling these systems with their cars. This project researches the use of four fish-eye lens cameras to produce a 360 bird's-eye view. The main approach for this revolves around rectifying the input images using camera calibration, aligning the four camera images to a common ground plane, and compositing them into a single image. A secondary approach was researched as well, which revolves around projecting the input images directly onto semi-spherical surfaces in a virtual environment to produce a composite image. The results from the main approach showed that it is possible to create a 360 view using only four fish-eye lens cameras. The resulting composited view is thought to be helpful to a driver, by covering areas that would be blinds-spots if only windows and mirrors were used.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.



Elektronik og IT
Aalborg Universitet
http://www.aau.dk

AALBORG UNIVERSITET
STUDENTERRAPPORT

Titel:
Creating 360 View by Combining Multiple Fish-eye Images

Tema:
Kandidatspeciale

Projektperiode:
Forårssemestret 2016

Projektgruppe:
16gr1045

Deltager(e):
Freddy Gramstrup Nielsen

Vejleder(e):
Prof. Thomas B. Moeslund
Aalborg University
Dr. Salvatore Livatino
University of Hertfordshire

Oplagstal: 1

Sidetal: 64

Afleveringsdato:
2. juni 2016

Abstract:

At sammensætte billeder fra flere kameraer monteret på en bil til et fugleperspektiv for at assistere en chauffør i at parkere er en relativt ny ide. Det er blevet forsket tidligere, men det er først for nyligt at bilproducenter er begyndt at sælge implementeringer af det med deres biler. Dette projekt forsker i brugen af fire kameraer med fish-eye linser til at producere et 360 fugleperspektiv. Den primære fremgangsmåde er baseret på at rette input billederne ved at bruge kamera kalibrering, justere de fire billeder til et fælles plan og sammensætte til et enkelt billede. Den anden fremgangsmåde er baseret på at projektere input billederne direkte på semi-sfæriske overflader i et virtuelt miljø for at skabe det endelige billede. Resultaterne fra den primære fremgangsmåde viste at det er muligt at skabe et 360 fugleperspektiv kun ved brug af fire kameraer med fish-eye linser. Det resulterende sammensatte billede menes at være nyttig for en chauffør, da det dækker områder der ellers ville være blinde vinkler hvis man kun brugte vinduer og spejle.

Rapportens indhold er frit tilgængeligt, men offentliggørelse (med kildeangivelse) må kun ske efter aftale med forfatterne.

Contents

Preface **ix**

Glossary **xi**

1 Introduction **1**
1.1 Problem Description 2
1.2 System Overview 3
1.3 Project Delimitation 4

2 Background Knowledge **5**
2.1 Fish-eye Lenses 5
2.2 Camera Calibration 6
2.2.1 World Coordinates 14
2.3 Image Stitching 17
2.3.1 Scale-Invariant Feature Transform 18
2.3.2 Image Blending 22
2.4 Projection 24

3 State of the Art **27**
3.1 Bird's-Eye View Vision System for Vehicle Surrounding Monitoring 27
3.2 A Surround View Camera Solution for Embedded Systems 29
3.3 Nissan Around View Monitor 30
3.4 Discussion 31

4 Design and Implementation **33**
4.1 Approach 1: Planar Alignment and Composite Generation 34
4.1.1 Design 34
4.1.2 Implementation 35
4.1.3 Discussion 41
4.2 Approach 2: Direct Spherical Projection using Unity 42
4.2.1 Design 42

4.2.2 Implementation 42
4.2.3 Discussion 45

5 Results **47**
5.1 Test Scenario 1 47
5.2 Test Scenario 2 48
5.3 Test Scenario 3 49
5.4 Test Scenario 4 51
5.5 Test Scenario 5 52
5.6 Test Scenario 6 53
5.7 Test Scenario 7 54
5.8 Comparison to Related Work 55

6 Discussion **59**
6.1 Future Work 59
6.2 Conclusion 60

Bibliography **61**

A Image Acquisition **63**

Preface

This project was done while abroad at University of Hertfordshire in Hatfield, UK. The project is a continuation of work done on a hardware set-up provided by STMicroelectronics with four fish-eye lens cameras streaming via a UDP protocol. The work is done by a computer engineering student and focuses heavily on Computer Vision approaches. The target reader is therefore someone with an interest in that field and who has some background knowledge beforehand, though more technical aspects will be explained throughout the report.

Acknowledgements

I would like to take this opportunity to thank my supervisors on this project Professor Thomas B. Moeslund (AAU) and Dr. Salvatore Livatino (UH) for constructive feedback, and guidance in both report writing and project direction in general. I would also like to thank STMicroelectronics who provided the test set-up. And finally a thanks to Madhav Bhaku, who worked with this system earlier, for giving me a quick introduction of how to work with the cameras on the hardware set-up.

Aalborg University, June 02, 2016

Freddy Gramstrup Nielsen
<fniel11@student.aau.dk>

Glossary

BBF	Best-Bin-First
CG	Computer Graphics
CV	Computer Vision
DoG	Difference of Gaussians
DIW	Dynamic Image Warping
FOV	Field Of View
GPU	Graphics Processing Unit
IP	Internet Protocol
RAM	Random-Access Memory
RGB	Red-Green-Blue
SIFT	Scale-Invariant Feature Transform
UDP	User Datagram Protocol

Chapter 1

Introduction

For the purpose of making cars and other vehicles more secure for the drivers and pedestrians, in the past rear-view cameras have been added especially to large cars and trucks to let the driver see areas that would be blocked otherwise. More recently some car manufacturing companies (and third-parties) have developed systems that gives the driver the opportunity to see most if not all the cars surroundings, even in traditional blind-spot areas.

There are different ways of covering a large area with cameras. You could have many regular cameras and combine/stitch their images together into one. You have a single moving camera and combine the images taken at different angles. Finally you could have a few wide angle lens cameras that are able to cover the entire 360 view. For the use case of vehicles it would not be practical to use neither many regular cameras or a single moving camera, as it would be both expensive and difficult to cover the entire wanted area.

The problem with using few cameras with wide angle lenses is that the lens distorts the image a lot in order to get such a wide view (distortion is worse the further an object is from the optical centre in the image). Therefore camera calibration is needed to undistort the images taken with each camera.

The goal of this project is to create a bird's-eye view of the test set-up provided by STMicroelectronics, which has four fish-eye lens cameras mounted on it. The four individual images should be combined into a single image, covering the full 360 degrees surrounding the car.

Different solutions were investigated throughout the project's period and in the end a main approach was chosen along with a secondary approach.

This project continues the work on the 360CarView project done in [1].

In Chapter 2 the background knowledge needed in order to understand all aspects of the following chapters is explained, in Chapter 3 the State of the Art relevant to this project will be examined, in Chapter 4 the main approach and secondary approach will be explained, both in idea and implementation, in Chapter 5 the results from the two approaches is shown and discussed, along with a comparison to the results of other similar projects, and finally in Chapter 6 the results and project in general will be discussed and concluded upon.

1.1 Problem Description

In this section some of the problems that the proposed solution would fix are explained.

As explained in detail in [7], a driver in a car can not see everything around him/her. This is because parts of the car's frame is blocking the driver's view. All cars are required to have mirrors that help prevent some of this by letting the driver see behind the car and to the sides, but not all areas are covered.

Especially behind the car, the back hatch covers up a large region making it difficult if not impossible to see some objects, pets, or most importantly people and children. An example of the blind-zones for a car can be seen in Figure 1.1 including the area behind the back hatch. The rear-view blind-zone is dependent on the driver's height, in [7] they measured how far back a 0.71m tall traffic cone needed to be for a driver to see it. In the same four-wheel drive vehicle, for a 1.73m tall driver the distance was 13.4m, while for a 1.55m driver the distance was 21m.

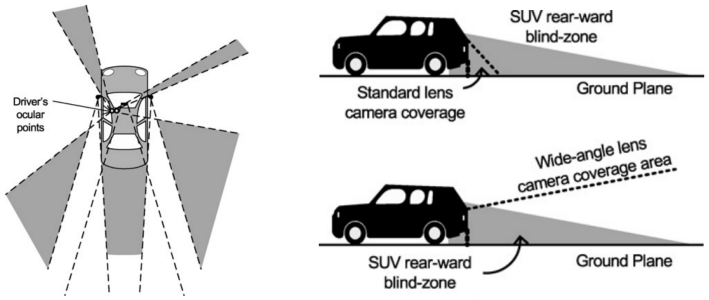


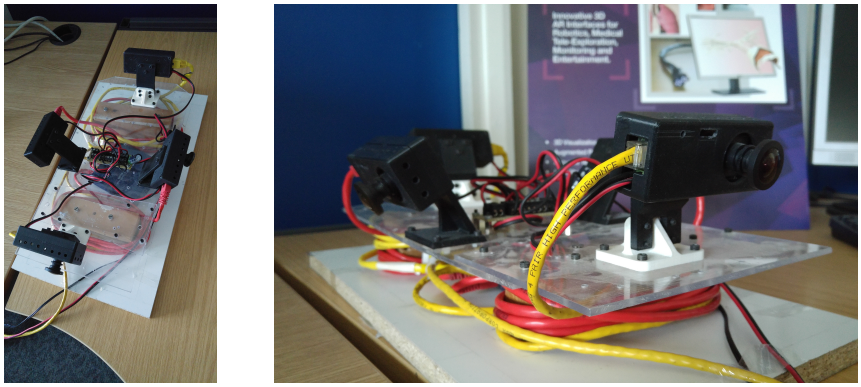
Figure 1.1: Images showing the blind-zones of a car, (a) top-down and (b) side-view also showing how using a wide-angle lens camera could help the driver gain more visibility. [7]

The goal of this project is to create a 360 bird's-eye view of the cars surroundings in order to assist a driver in a parking scenario, both to make it easier to park, and

more importantly to make it safer both for driver and his/her surroundings as the driver can see if something or someone is in the way of the vehicle.

1.2 System Overview

The system used for this project has been supplied by STMicroelectronics, it is a rigid set-up with four cameras mounted as seen in Figure 1.2, the size of the set-up is close to that of a large RC car, and can be seen as a small scale version of an actual car.



(a) (b)

Figure 1.2: The cameras mounted on the set-up, (a) top-down making it possible to see the camera positions, and (b) side view making it easier to see the different angles at which the cameras are pointing downwards.

The four cameras are pointing towards front, rear, left, and right (90 degrees apart) and are pointing slightly downwards with different angles. The cameras are somewhat low in resolution with 512x512, and because they use circular fish-eye lenses, only the inner circle contains information (making the total usable pixel count much lower). The cameras stream images through UDP, how this works can be found in Appendix A. The set-up is powered by an external power-supply running at around 13.5V which is in the area of what a car battery will output, which makes sense given the fact that the goal is that this can be used for cars.

A 5-port ethernet switch is used to connect the four cameras with a PC.

1.3 Project Delimitation

Most of the delimitations of the project comes from the supplied hardware set-up, which is described in Section 1.2.

Even though they can lead to complications, this project addresses the use of fish-eye (ultra wide angle) lenses, this is what is wanted from the company who supplied all the hardware (STMicroelectronics). Their goal is to create a surround view using as few cameras as possible while still maintaining a usable result.

This project will focus mainly on what is possible with only four fish-eye lens cameras, as that is what is present on the provided set-up for the project.

The project will be focused on creating a top-down view of the car, as this is seen as the most helpful view for the user when parking. Adding a view from the front or rear camera might be useful as well, but that would be a trivial extra to implement, and was therefore chosen not to focus on.

Chapter 2

Background Knowledge

In this chapter the theory deemed necessary for the reader to properly understand the later chapters will be explained. This is general theory and later implementations might deviate slightly from this.

2.1 Fish-eye Lenses

In this section the lenses equipped on the cameras used for this project will be explained. Most cameras use lenses which produce images that will look natural under normal conditions when viewed by a human, this means for example that straight lines look straight, this is done by having the focal length be around the same length as the diagonal of the sensor in the camera.



(a) **(b)**

Figure 2.1: Two images taken from the same position using (a) a standard camera lens from smartphone, and (b) one of the fish-eye lens cameras used in the project.

The lenses used for this project however is what is called fish-eye lenses, which have ultra wide viewing angles. There are two main types of fish-eye lenses, full

frame which contains no borders, but has a lower Field of View (FOV) in the vertical direction than the horizontal direction. The other type is called circular fish-eye lenses which is what is used for this project, they are mapped spherically making the image "round" in contrast to standard rectangular images. An example showing the differences between a normal lens and a circular fish-eye lens can be seen in Figure 2.1.

Fish-eye lenses are not often used for general photography, but has its applications where a wide angle is needed and the cost should be effective, not that fish-eye lenses are cheap as such but the alternative would be multiple cameras which would in most cases be much more expensive. Examples of usages is meteorology for monitoring cloud formations, or in video surveillance in order to cover a larger area.

The wide angle is obtained by using multiple lenses of different shapes, an example can be seen in Figure 2.2. This illustration shows how the light bends from one lens to the other until finally reaching the sensor.

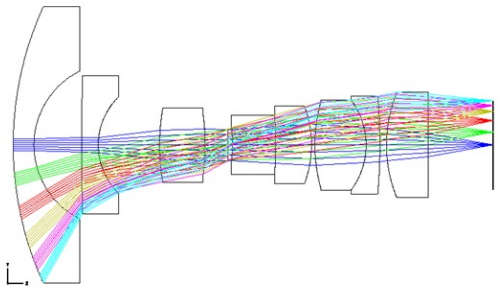


Figure 2.2: An example of how a fish-eye lens is constructed, and how the light bends through the individual lenses. [16]

The wide angle lens causes straight lines to bend. This effect is called barrel distortion and is worse the further from the optical centre an object is, which can be seen from the fish-eye lens example in Figure 2.2. Camera calibration is a way of rectifying this effect, which is explained in Section 2.2.

2.2 Camera Calibration

In most Computer Vision (CV) related subjects it is important that there is as little distortion as possible in the images that are being processed. Because the images from the fish-eye cameras used in this project are heavily distorted, it is important to rectify them as best as possible. There are many different ways of doing this, the simplest approaches attempt to stretch the image by different factors, stretching

the outer edges more than the centre area, approaches like this can be useful in some cases, but because it only "guesses" how to produce an output, a more advanced approach is more commonly used for this. Camera Calibration attempts to model the camera's lens and sensor parameters (intrinsics) along with the camera's position and viewing angle (extrinsics). The intrinsics are camera and lens dependent, and therefore there is only one set of intrinsic parameters for each camera, whereas there is a set of extrinsic parameters for each image. The main source of information used for this Section is [2] and to a lesser degree [18].

The most common type of lens distortion is radial distortion which is very prominent in the cameras with wide viewing angles such as the ones used in this project. Radial distortion causes things to look as if they curve outwards from the optical centre of the image and this effect will increase the further away from the centre it is. An illustration of the cause of radial distortion can be seen in Figure 2.3(a). The second most common type of distortion is called tangential distortion, which occurs because of faults in manufacturing if the lens is not precisely parallel to the imaging sensor. An illustration of the cause of tangential distortion can be seen in Figure 2.3(b).

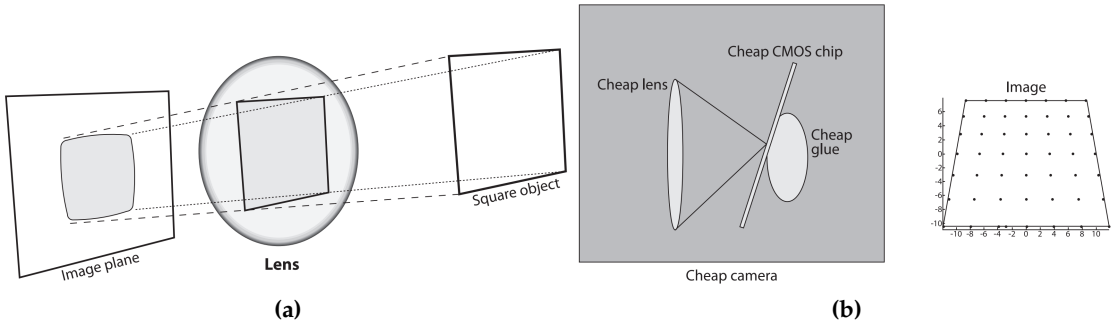


Figure 2.3: The two most common types of lens distortion, (a) radial and (b) tangential [2].

As this project will contain several Computer Vision (CV) methods in order to get the wanted final product of a bird's-eye view, it is vital to rectify the image, and remove the severe radial distortion that fish-eye lenses produce. The distortion changes how objects look depending on how far from the optical centre they are, which makes for aligning the four images difficult, and if the image is not undistorted the feature matching part of image stitching will be poor.

There is no radial distortion in the optical centre of an image and it will increase with the distance to the optical centre. Standard lenses have a low amount of distortion which can in most cases be completely disregarded, but if camera calibration was deemed necessary, it can be characterized by just the two first terms of a Taylor series expansion around r = 0, these two terms are called k_1 and k_2, but

for lenses with higher amounts of distortion like the ones used in this project, the third term k_3 is also needed. Generally points can be corrected using the following two equations:

$$\begin{aligned} x_{corrected} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_{corrected} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{aligned} \tag{2.1}$$

Where x and y are the original coordinates on the image, and $x_{corrected}$ and $y_{corrected}$ are the corrected coordinates.

Tangential distortion can similarly be represented by terms of a Taylor series, the two terms are called p_1 and p_2, the correction of tangential distortion can be done as shown in the following equations.

$$\begin{aligned} x_{corrected} &= x + (2p_1 y + p_2(r^2 + 2x^2)) \\ y_{corrected} &= y + (p_1(r^2 + 2y^2) + 2p_2 x) \end{aligned} \tag{2.2}$$

This means that there are five coefficients that describe the general distortion (k_1, k_2, k_3, p_1, and p_2).

Along with these, the focal length of the camera (f_x,f_y) and the optical centre (c_x,c_y) has to be found as well. These are represented by a 3x3 matrix:

$$camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

This means that there are nine unknowns that needs to be found, this is what camera calibration does, and once they have been estimated the images can be undistorted/remapped using these parameters.

There are many different approaches to camera calibration, most of them involving presenting a known pattern to the camera. The approaches can in general be split up into four categories as described in [18].

- **3D reference object based calibration:** Here the used pattern is a 3D calibration object, which is presented to the camera and used to calibrate, normally consisting of three orthogonal planes. Requires expensive apparatus and set-up.

- **2D plane based calibration:** Perhaps the most used, here a single pattern plane is presented to the camera. A simple approach that does not require expensive materials, as anybody can print out their pattern of choice.
- **1D line based calibration:** Here the calibration object consists of a number of collinear points. Usually this is done with a stick containing three or more easily findable points with known distances between each other. A quite cumbersome approach to implement.
- **Self-calibration:** This approach is far more mathematically advanced, as there is no presented object. Instead the camera is moved in the scene and the parameters are estimated using only the image information. This is a less intuitive and more difficult approach to implement.

The calibration object chosen here is the 2D object, because it is the most standard way, and because the calibration object can simply be printed out (and put on a rigid surface). In this category there are different options for the pattern to be used, the most used is a chessboard or circle grid pattern, an example of both can be seen in Figure 2.4. For this project the chessboard pattern was chosen. To calibrate the camera, the pattern should be presented at various angles and distances to the camera.

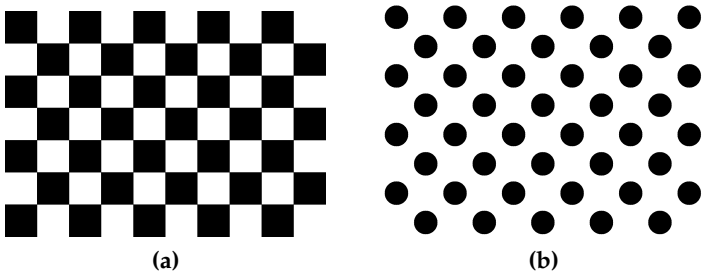


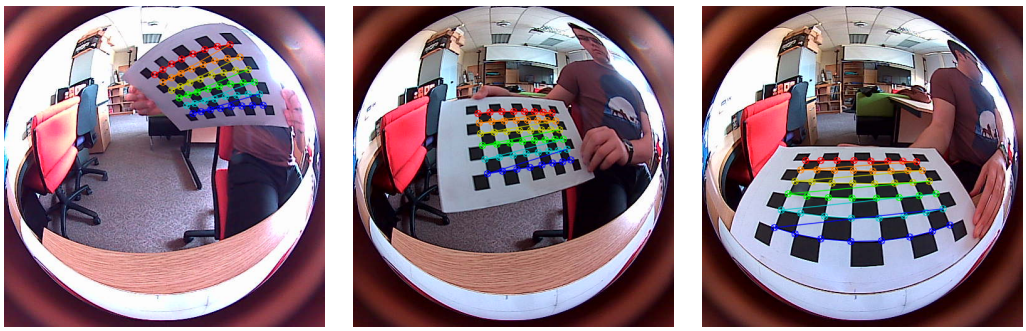
Figure 2.4: Examples of calibration patterns, (a) chessboard pattern and (b) circle grid pattern.

In each image one can then search for and find corners (intersections between the black squares) to get a set of 2D points for each image. Along with these image coordinates, a set of object coordinates in 3D space must be specified as well. For ease of calculation one can say that the z-value is always 0, to justify this it is important to note that the images can either be seen as a static camera taking images of a moving calibration object, or that the camera moves around the object. This would, with respect to the pattern, result in the same. Another thing to note is that the squares in the chessboard can be said to be 1 “unit” in width and height, this will output a camera matrix that also uses this “unit” and can easily be changed to whichever side length the squares have. From this you can generate a set of 3D points as follows:

$$
\begin{aligned}
&(0,0,0),(0,1,0),(0,2,0),\ldots,\\
&(1,0,0),(1,1,0),(1,2,0),\ldots,\\
&\ldots\\
&(B_{width}-1,B_{height}-1,0)
\end{aligned}
\tag{2.4}
$$

Where B_{width} and B_height are the number of intersections between squares across the width and height respectively (using the pattern in Figure 2.4(a) it is 8x6), by multiplying with the actual side length of the squares in cm, all the resulting output values will also be in cm.

From this two sets of coordinates have been found; 2D coordinates in the image (see Figure 2.5) , and 3D coordinates in the grid. These coordinates correspond to each other, it is important that the points are in the same order, so that the right points are used together.



(a) **(b)** **(c)**

Figure 2.5: Chessboard corners found in three images from the rear camera on the set-up.

For each view of the pattern, a homography matrix can be calculated, a homography can be used to map from one plane to another.

An equation can be created that describes an image point $\tilde{q}$:

$$\tilde{q} = sMW\tilde{Q} \tag{2.5}$$

Where $\tilde{q}$ is an observed point on the image plane, $\tilde{Q}$ is a point on the object plane, s is a scale factor, M is the camera matrix of intrinsic parameters, and $W = \begin{bmatrix} R & t \end{bmatrix}$ where R and t are the rotation matrix and translation vector for the current view.

Because all z values are 0 in the object coordinates, the third rotation vector of R can be disregarded:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = sM \begin{bmatrix} r_1 & r_2 & r_3 & t \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} = sM \begin{bmatrix} r_1 & r_2 & t \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.6}$$

From this the homography matrix H that maps from the object point plane to the image point plane can be described as $H = sM \begin{bmatrix} r_1 & r_2 & t \end{bmatrix}$, where $\tilde{q} = sH\tilde{Q}'$. Note that $\tilde{Q}'$ is only defined in the observed plane, whereas $\tilde{Q}$ is defined in all space.

The homography matrix H describes the relation between the points on source and destination image planes as seen in the following equations:

$$\begin{aligned} p_{dst} &= Hp_{src}, \quad p_{src} = H^{-1}p_{dst} \\ p_{dst} &= \begin{bmatrix} x_{dst} \\ y_{dst} \\ 1 \end{bmatrix}, \quad p_{src} = \begin{bmatrix} x_{src} \\ y_{src} \\ 1 \end{bmatrix} \end{aligned} \tag{2.7}$$

One of the things to consider when doing camera calibration, is how many images are needed for each camera. Technically it would be enough with only a couple of images [2], but it is recommended to use at least ten, because there could be noise in the images, and more images will reduce the effect of this.

If H is split into three 1x3 vectors we get the following equation:

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} = sM \begin{bmatrix} r_1 & r_2 & t \end{bmatrix} \tag{2.8}$$

from this, three separate equations can be extracted:

$$\begin{aligned} h_1 = sMr_1 \quad & or \quad r_1 = \lambda M^{-1}h_1 \\ h_2 = sMr_2 \quad & or \quad r_2 = \lambda M^{-1}h_2 \\ h_3 = sMt \quad & or \quad t = \lambda M^{-1}h_3 \end{aligned} \tag{2.9}$$

Where λ is $\frac{1}{s}$. The rotation vectors are orthogonal to each other, and because the scale factor has been separated they are orthonormal as well, meaning that their dot product is 0 and magnitudes are the same. This means that:

$$r_1^T r_2 = 0 \tag{2.10}$$

Using the rule that for any vectors a and b we have $(ab)^T = b^T a^T$, if r_1 and r_2 are substituted using what is stated in equation 2.9 the first constraint can be derived:

$$h_1^T (M^{-1})^T M^{-1} h_2 = 0 \tag{2.11}$$

The magnitudes of the two rotation vectors are known to be equal, meaning:

$$||r_1|| = ||r_2||, \quad or \quad r_1^T r_1 = r_2^T r_2 \tag{2.12}$$

Again, substituting r_1 and r_2 the second constraint can be derived:

$$h_1^T (M^{-1})^T M^{-1} h_1 = h_2^T (M^{-1})^T M^{-1} h_2 \tag{2.13}$$

$(M^{-1})^T M^{-1}$ is set to B, which have a general closed-form solution which is:

$$B = (M^{-1})^T M^{-1} = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix} = \begin{bmatrix} \frac{1}{f_x^2} & 0 & \frac{-c_x}{f_x^2} \\ 0 & \frac{1}{f_y^2} & \frac{-c_y}{f_y^2} \\ \frac{-c_x}{f_x^2} & \frac{-c_y}{f_y^2} & \frac{c_x^2}{f_x^2} + \frac{c_y^2}{f_y^2} + 1 \end{bmatrix} \tag{2.14}$$

If B substitutes $(M^{-1})^T M^{-1}$ in the constraints, it can be seen that both constraints are of the same general form $h_i^T B h_j$. It can be seen in 2.14 that B is symmetric, which means that the unique values can be taken out to form a six dimensional vector b:

$$h_i^T B h_j = v_{ij}^T b = \begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{bmatrix}^T \begin{bmatrix} B_{11} \\ B_{12} \\ B_{22} \\ B_{13} \\ B_{23} \\ B_{33} \end{bmatrix}^T \tag{2.15}$$

When K images of the pattern has been acquired, V_{ij}^T is a 2Kx6 matrix, if $K \geq 2$ there is a solution (though ten or more images will be better and more robust to noise). From the values of B the intrinsic parameters can be extracted using the following equations:

$$\begin{aligned} f_x &= \sqrt{\frac{\lambda}{B_{11}}} \\ f_y &= \sqrt{\frac{\lambda B_{11}}{B_{11}B_{22} - B_{12}^2}} \\ c_x &= -\frac{B_{13}f_x^2}{\lambda} \\ c_y &= \frac{B_{12}B_{13} - B_{11}B_{23}}{B_{11}B_{22} - B_{12}^2} \end{aligned} \tag{2.16}$$

Where $\lambda = B_{33} - \frac{B_{13}^2 + c_y(B_{12}B_{13} - B_{11}B_{23})}{B_{11}}$.

Then the extrinsic parameters can be computed using the the equations in 2.9:

$$\begin{aligned} r_1 &= \lambda M^{-1}h_1 \\ r_2 &= \lambda M^{-1}h_2 \\ t &= \lambda M^{-1}h_3 \end{aligned} \tag{2.17}$$

The third rotation vector can, because the vectors are orthogonal, be calculated simply by taking the crossproduct between the two others $r_3 = r_1 \times r_2$.

The final part of the camera calibration procedure is to calibrate the distortion coefficients, as mentioned earlier there are three for radial distortion (k_1, k_2, and k_3), and two for tangential distortion (p_1 and p_2).

The observed points in an image are because of distortion in the wrong location. Let (x_p,y_p) be the location of a point if the camera was perfect, and (x_d,y_d) be that point's distorted location.

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = \begin{bmatrix} f_x \frac{X^W}{Z^W} + c_x \\ f_y \frac{X^W}{Z^W} + c_y \end{bmatrix} \tag{2.18}$$

Using the results of the camera calibration, to substitute into:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{bmatrix} x_d \\ y_d \end{bmatrix} + \begin{bmatrix} 2p_1 x_d y_d + p_2(r^2 + 2x_d^2) \\ p_1(r^2 + 2y_d^2) + 2p_2 x_d y_d \end{bmatrix} \tag{2.19}$$

This will produce a large number of equations (because of all the input images) to be solved in order to find the distortion coefficients.

2.2.1 World Coordinates

The previously mentioned calibration deals with the camera itself and its lens. However because this is a multi-camera set-up, it could be helpful to use extrinsic parameters to find the cameras' positions in the real world as well, which describes the rotation and translation (position) of the camera in world space. For this some assumptions/definitions must be set up, some assumptions of the camera is that they are pointing 90° apart, and they are placed in the world coordinate system as seen in Figure 2.6.

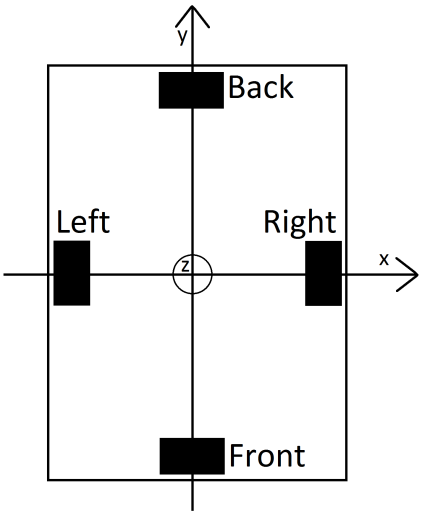


Figure 2.6: World coordinate system, note that the z-axis is pointing upwards according to the right-hand rule.

From this it can be said that the yaw angle (R_x) is equal to the values shown in Table 2.1.

Moreover it is assumed that all four cameras have a roll angle of 0°, if they were different from zero the cameras would be tilted. Then what is left to be found is the pitch angle, which must be calculated.

Camera	Yaw angle
Back	0°
Right	90°
Front	180°
Left	270°

Table 2.1: Yaw angles of the four cameras.

The approach for finding this is that if a calibration pattern is placed in front of the camera lying flat on the ground/surface, its yaw and roll angles are 0° and its pitch angle is 90°. So in the calibration procedure, a rotation and translation vector can be estimated which describes how the pattern is oriented in the image, or if you think of it the other way around, how the camera is positioned in order to get the image at that angle.

As mentioned earlier the object coordinates are all defined as zero on the z-axis, in the calibration these coordinates are seen as the "world coordinates", lets call them object coordinates though, to not confuse them with the world coordinates as shown in Figure 2.6. So the rotation and translation vectors from each view describes the camera position with respect to the object world coordinate system.

So by physically measuring the distance in the x-axis and y-axis from the object world coordinate origin to the global world coordinate origin (the origin in object world is the top left corner of the calibration pattern). For example the position in world coordinates of the object world coordinate system's origin for the front camera was measured as being (7.2, -44.3, 0) measured in cm, again for simplicity the z value is set to zero as this makes the surface/floor surrounding the set-up all zero in the z-axis.

Then if the rotation vector is transformed into a standard 3x3 rotation matrix using for example Rodrigues' formula, the position of the camera in object coordinates can be found by:

$$camera\ position = (R^T)^{-1} * t \tag{2.20}$$

Where R is the rotation matrix and t is the translation vector. This gives (for the front camera) (7.38; 18.72; -11.06) in the object coordinates. And given that the position of the origin is known the position of the cameras can be approximated. Though the calculation will be slightly different for each of the cameras as the object coordinate systems are all rotated differently. This can be seen in Figure 2.7.

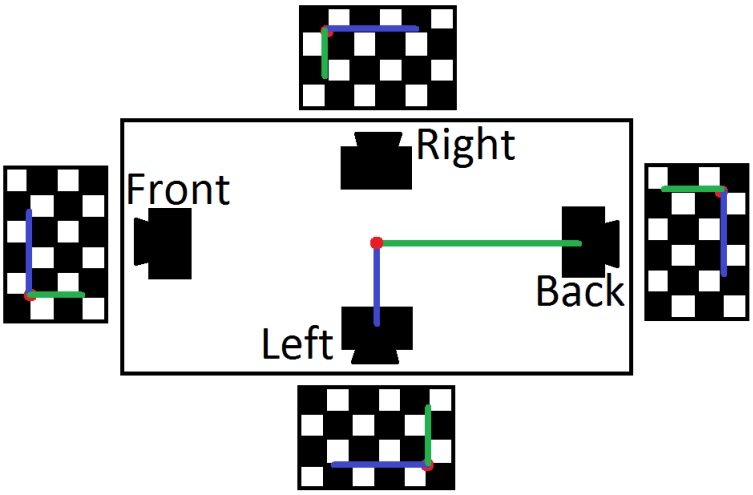


Figure 2.7: Top-down view of the object coordinate systems placement in the world coordinates (note that according to the right-hand rule, except world coordinates, all z-axes point downwards).

From this it can be seen that the axis positions of the front camera can be calculated as:

$$\begin{aligned} front_cam_world_x &= origin_object_x - front_cam_object_x \\ front_cam_world_y &= origin_object_y + front_cam_object_y \\ front_cam_world_z &= origin_object_z - front_cam_object_z \end{aligned} \tag{2.21}$$

Whereas the equations change a bit when for example the left camera axis positions are calculated, as it is rotated by 90 degrees (so the x-axis of one corresponds to the y-axis of the other and vice versa):

$$\begin{aligned} left_cam_world_x &= origin_object_x - left_cam_object_y \\ left_cam_world_y &= origin_object_y - left_cam_object_x \\ left_cam_world_z &= origin_object_z - left_cam_object_z \end{aligned} \tag{2.22}$$

Now all positions are known, the next step is to extract the rotation/pose of the camera. This can be done using the rotation matrix. By itself this does describe the rotation of the calibration object in the camera view (as mentioned earlier), so that means that the matrix should be transposed in order to explain the rotation of the camera according to the calibration object.

In [5] it is explained how to extract euler angles from a rotation matrix, this can often result in either imprecise results or issues with *Gimbal Lock* which means that for example the second rotation could align the first and third making the rotations not unique, this method attempts to avoid this by first calculating the first and second rotations and then calculate what the third angle should be in order to match the target rotation (from the matrix). Given the rotation matrix:

$$M = \begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & M_{22} \end{bmatrix} \tag{2.23}$$

The equations needed to calculate the euler angles are:

$$\begin{aligned} \theta_1 &= atan2(m_{12}, m_{22}) \\ c_2 &= \sqrt{m_{00}^2 + m_{01}^2} \\ \theta_2 &= atan2(-m_{02}, c_2) \\ s_1 &= sin(\theta_1) \\ c_1 &= cos(\theta_1) \\ \theta_3 &= atan2(s_1 m20 - c_1 m_{10}, c_1 m_{11} - s_1 m_{21}) \end{aligned} \tag{2.24}$$

Where θ_1, θ_2, and θ_3 are the three rotations, which of cause can easily be calculated from radians to degrees.

Thereby all the information that is wanted about the cameras have been found, the intrinsic parameters: focal length, optical centre, and distortion coefficients along with extrinsic parameters: the position of the camera in world-space and its pose/pointing direction.

2.3 Image Stitching

In many applications it is useful to use multiple images to create a single wider view composite image, this is called stitching.

The general approach of image stitching is to find the same part of the same object in both camera's images, and then align those with each other, doing this will result in having an area with redundant data, here an approach is needed to choose what the pixel values of the resulting image should be. There are many ways of doing this, most are variations on the two listed here.

- **Superimposing:** Align the matching interest points but only use the pixel values from one of the cameras in the overlapping region. This will in most cases produce poor results on the boundary between the region and the second image which was not used in the output of the overlapping region. This boundary could be minimized by smoothing the transition between the two images.
- **Blending:** Align the matching interest points and blend the two images in some way. The blending is always done using x of image 1 and 1.0-x of image 2, sometimes x is set to 0.5, sometimes it is useful to interpolate, so that it increase/decrease from one image to the other in order to have a smooth transition.

Both of these compositing methods will be investigated.

2.3.1 Scale-Invariant Feature Transform

The goal of SIFT is to find stable and strong features, meaning that they should be robust with respects to scale differences, rotations, and illumination changes. There are many steps to go through to create a SIFT scale space, which will be explained in this section.

Octave Generation

Firstly as the features need to be scale invariant, a scale space has to be created. This is done by downscaling the image a number of times, for example by halving the size of the image. This is however not the only way the SIFT scale space is created, a second scaling is done by the use of blurring using Gaussian kernels. A number of different Gaussian kernels are being applied to each image in the original scale space. an image and all its blurred versions is called an octave, where the first octave is the original image and all of its blurred counterparts, the second octave is the first downscaled image and its blurred versions and so on. This can be seen in Figure 2.8.

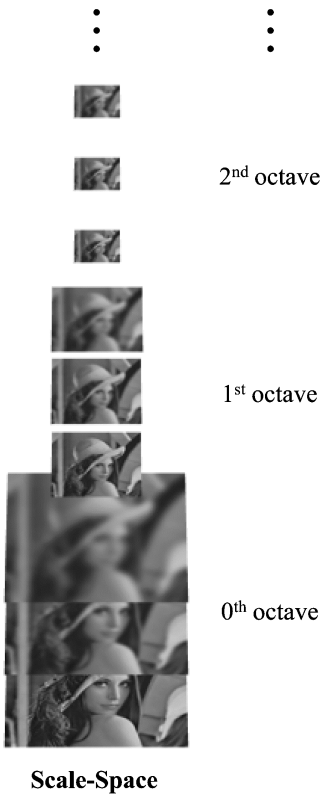


Figure 2.8: How the SIFT algorithm creates the octaves in the scale space. [9]

The equations used to blur/smooth the images in the octave is:

$$\begin{aligned} L(x,y,\sigma) &= G(x,y,\sigma) * I(x,y) \\ G(x,y,\sigma) &= \frac{1}{2\pi\sigma^2} e^{\frac{x^2+y^2}{2\sigma^2}} \end{aligned} \tag{2.25}$$

Where I is the original image, L is an image with Gaussian blur applied, G is the Gaussian operator, x and y are image coordinates, and σ is the scale of the blurring, so a greater value of σ would mean a more blurred image L. The number of octaves and scales per octave needed depends on the original image resolution.

Keypoint Localization

When all the octaves have been created, keypoints can be extracted in each octave. This is done using Difference of Gaussians (DoG) which means that image 1 and 2 are subtracted from one another, image 2 and 3 and so on, this can be seen in

Figure 2.9(a). Then the DoGs are searched for extrema, if a point is the maximum or minimum within its 26 neighbourhood pixels (see Figure 2.9)(b)) it is considered an extrema and used as a keypoint (some filtering is done to remove edge-points).

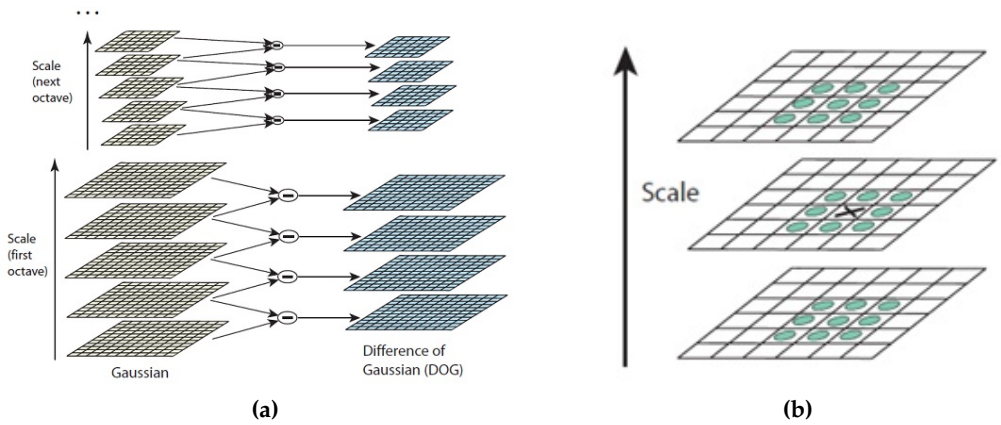
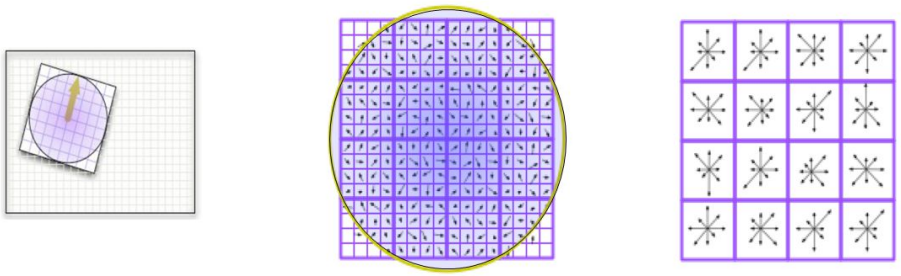


Figure 2.9: SIFT keypoint finding, (a) the principles of DoG and (b) maxima localization. [15]

After localizing these extrema keypoints, a dominant orientation is assigned to achieve invariance to rotations. This is done by taking a neighbourhood surrounding the keypoint (the size differs depending on the scale), wherein the gradient direction and magnitude is calculated. A 36-bin histogram is created covering the total 360 degrees. The histogram is weighted by the gradient magnitude, and the highest peak is used as the dominant orientation (some approaches involves taking the orientations that are above 80% of the peak to create multiple keypoints that share location and scale, but has a different rotation).

Generating SIFT Features

Then the keypoint descriptors are created. This is done by using a neighbourhood region of 16x16 around the keypoint, so each of them contains a 4x4 subregion. In each subregion the gradients and their magnitudes are collected into an 8-bin histogram (points closer to the keypoint are weighted higher), resulting in 16 subregions each having 8 histogram values, which makes for a 128-dimensional feature for each keypoint, a simplified representation of the SIFT feature generation can be seen in Figure 2.10.



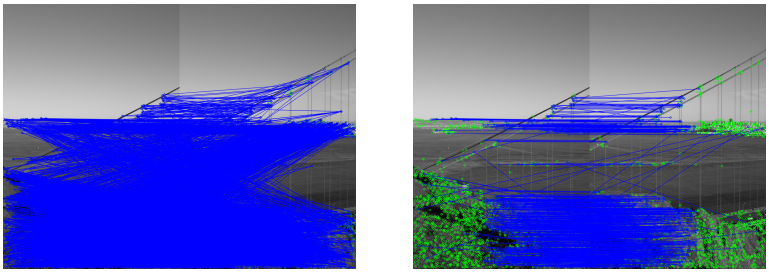
(a) (b) (c)

Figure 2.10: The stages of SIFT feature generation, (a) finding the dominant direction, (b) finding 16x16 pixels around keypoint, and (c) putting this into histogram-bins to create the 128-dimensional feature vector. [12]

SIFT Matching

Having produced feature vectors for two images, the next approach is to search for matches between them. There are different ways of doing it, either brute-force which can be inefficient with a large number of features, or as described in Lowe's original SIFT paper [11], a neighbourhood search. The search method used in [11] is called Best-Bin-First (BBF) which searches for the closest neighbour with a high probability of being a match.

After that it is important to filter out bad matches. This can be done using a ratio test, if the ratio between the best match and the second-best match is below a threshold the match can be rejected, in Lowe's paper [11] he suggests using 0.8 as the threshold, as he found that it filtered out around 90% of the incorrect matches, while only disregarding 5% of the correct matches. This can be seen in Figure 2.11, where matches between two overlapping images are shown before and after ratio test.



(a) (b)

Figure 2.11: Sift matching, (a) all points matched, and (b) points matched after ratio test with threshold set to 0.8.

2.3.2 Image Blending

With features matched between two images, they can be stitched together into one composite image. The first step needed is to map one image to the other using the matched points to create a homography matrix. The homography matrix should be formed so that a point from the source image can be mapped to the destination image (and vice versa):

$$p_{dst} = Hp_{src}, \quad p_{src} = H^{-1}p_{dst} \tag{2.26}$$

Once one of the images has been warped to the other, they can be blended together. The simplest way it to average the two images this way:

$$P_{stitched} = 0.5 * P_{image1} + 0.5 * P_{image2} \tag{2.27}$$

This would however result in the pixels that don't overlap only have half the possible values (can only go between 0 and 127) while in the overlapping region the resulting pixel are influenced by both and therefore have a full range (0 to 255). A result of this can be seen in Figure 2.12.



(a) **(b)** **(c)**

Figure 2.12: Image blending, (a) image 1, (b) image 2, and (c) average blending.

A better approach would be to only average blend where the images overlap as seen in Figure 2.13.

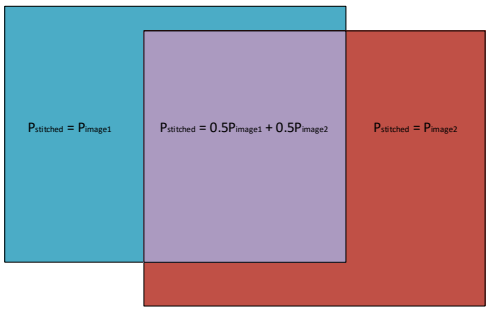


Figure 2.13: Blending is done differently for the three areas, averaging where there is overlap

The result of using this approach can be seen in Figure 2.14.



(a) **(b)** **(c)**

Figure 2.14: Image blending, (a) image 1, (b) image 2, and (c) average blending.

There is still slightly visible borders in the stitched image though, and there are ways of blending that is better at reducing that by dynamically choosing the weight of the blending (instead of having it always be 0.5).

This way images can be stitched together to form a single composite image. Some issues might arise during implementation of this, due to the cameras being positioned differently and therefore objects in view will look vastly different making it difficult to match features.

2.4 Projection

Image projection is a Computer Graphics (CG) method that maps a flat image onto a surface, it can also happen the other way around, though that is not relevant to this project.

If a panorama that covers 360x180 degrees has been made (some of it can be empty, for example the top half would be mostly irrelevant to the driver in a parking situation) there are different ways of making use of it. From an implementation standpoint it would be easy to just show the panorama itself to the user, but this would be confusing and unhelpful. Another idea is to use projection, which is what this section focuses on. There are three main projection methods, though others do exist, these three can be seen in Figure 2.15.



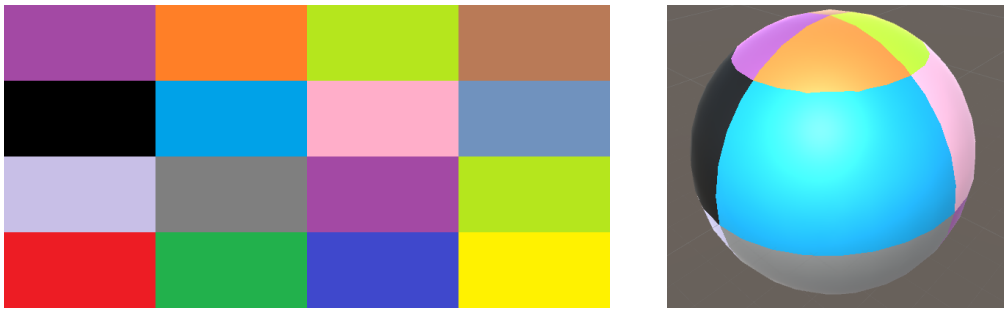
(a) (b) (c)

Figure 2.15: A globe as projected by, (a) planar, (b) cylindrical, and (c) spherical projection. [8]

As is evident from Figure 2.15, the planar projection is distorted in both the horizontal and vertical directions (increasing with the distance to the centre), the cylindrical projection is distorted in the vertical direction, while spherical projection is mostly without distortion.

Because the bird's-eye view is the most wanted view for this project, it will make most sense to create a spherical projection. This way the bird's-eye view can easily be created by moving the virtual camera above the sphere and pointing it downwards. The cameras do not cover the full 360x180 area, as there are parts above and below without information. A way to remedy this is to place a car in the virtual scene where a car would be if this was a real set-up, the missing information above the car can be neglected as it is never relevant for the user to see. Multiple virtual cameras can be placed together making it possible for the user to for example both see a bird's-eye view and the rear camera when he/she is backing the car.

An example of how the Unity3D program applies an image to a sphere, can be seen in Figure 2.16, where a 2048x1024 image has been applied to a sphere.

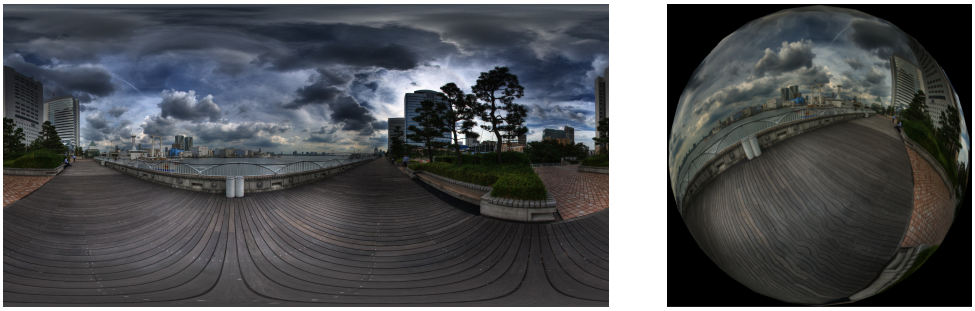


(a) (b)

Figure 2.16: Projection in Unity, (a) panorama image (b) applied as texture to a sphere (right).

So if there are missing parts of the 360 view it would be helpful to pad the missing areas, so the image does not get too stretched, this can be done by making sure the image is in 2:1 format, as the entire 360° are shown horizontally, and 180° vertically.

A 360x180 was found and applied to a sphere, this can be seen in Figure 2.17.



(a) (b)

Figure 2.17: Projection and viewpoint generation, (a) original panorama, (b) applied to a sphere with flipped normals. [4]

With the image projected onto the inside of a sphere (by flipping the surface normals) it is easy to create different views of the scene by simply moving and rotating the virtual camera in the scene, examples of this can be seen in Figure 2.18.

It can be seen in Figure 2.18(a) that this panorama is not stitched perfectly in the lower part, but this would be the area where the virtual car would be placed and so would not matter, as the rest looks quite good.



(a) (b) (c)

Figure 2.18: Projection and viewpoint generation, (a) top-down view in the scene, (b) rear view in scene, and (c) front view in scene. [4]

It is impossible to create anything as close to the quality of this panorama with a set-up like the one used in this project (low resolution, distorted images, etc.). But it can be seen here, that if a panorama was created, it would be easy to generate the wanted views as seen in Figure 2.18.

Chapter 3

State of the Art

In this chapter the previous work done in similar areas and projects with similar goals will be examined and explained. The related projects have been chosen because of their good results, and interesting use of methods.

In [18] the standard camera calibration approaches are explained. In [2] the maths behind OpenCV's camera calibration functionalities are explained. In [19] they undistort fish-eye lens images using two pairs of vanishing points from a single image of a chessboard calibration pattern to estimate an initial guess on the camera parameters, followed by a minimization problem to increase the precision.

In [6] they stitch a 360 degree video using four fish-eye cameras. The result is a cylindrical projection, where they reduce the effects of ghosts by using seam estimation. In [3] they made a dynamic stitching program that could automatically stitch any number of images, no matter which order they were fed to the system, or if some of the images did not match, or if there are multiple stitching "scenes" in the batch of images.

3.1 Bird's-Eye View Vision System for Vehicle Surrounding Monitoring

In this paper [10], they have the same goal as this project which is to create a bird's-eye view for use in cars and other vehicles. Their set-up is different though, as they have six fish-eye lens cameras mounted on a real to-scale car.

The approach they use is firstly to calibrate all the cameras, and undistort them to get perspective images instead of the input fish-eye images. The next step is to do planar alignment to warp the images from their original view to a bird's-eye view.

When objects that are not coplanar to the ground surface are present, they get heavily distorted when the perspective is changed. When one of these objects are placed so that it crosses between two images, it will be misaligned as can be seen in Figure 3.1(d).

To fix this and to find out which input image should contribute to each pixel in the composite image a seam is estimated between each neighbouring pair of images. To avoid ghosting and blurring along the seams, Dynamic Image Warping (DIW) is used to move the images so that one side of the seam align as best as possible to the other side.

To remove the effect of images having different exposures, a weighted blending approach is used to stitch the images along their seams.

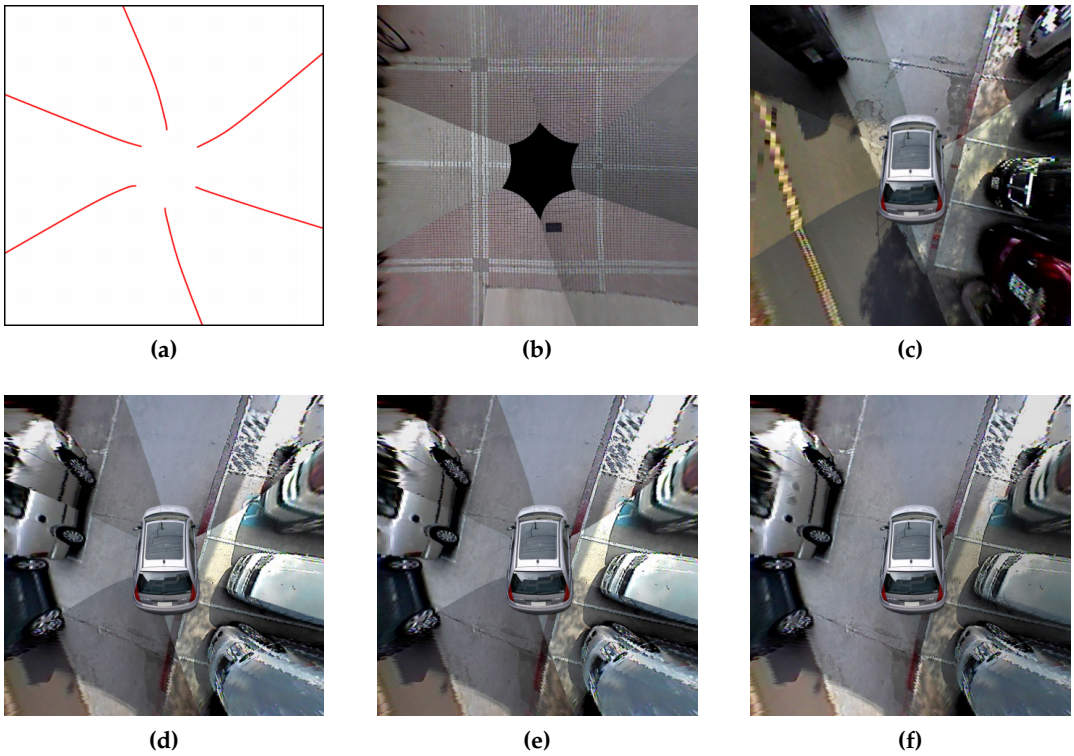


Figure 3.1: Approach and results from [10], (a) optimal seams, (b) composite image of the field for registration, (c) composite of ground-level objects, (d) composite of non-ground-level objects, (e) composite after registration along seams, (f) composite after compensating exposure and blending.

The advantage of this system is that the end result looks really good, with near perfect alignment because of the Dynamic Image Warping (DIW) applied along the seams between each neighbouring pair of images.

The disadvantages of this project is that it uses six cameras, instead of four, which

is the minimum (a single 360 fish-eye camera is not feasible in a vehicular scenario, as there is no single position to place it on the car that would cover all the wanted areas). This increases the price of it as a product. The system makes use of many different approaches such as seam estimation, non-ground object detection, DIW, blending, exposure compensation, etc. which might result in it being computationally heavy and will need more processing power to run in real-time.

3.2 A Surround View Camera Solution for Embedded Systems

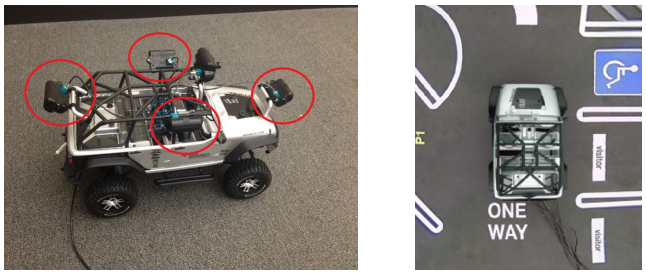
The approach in this paper is similar to that mentioned in Section 3.1. A focus that is different in this approach is that they implement it on an embedded system. The approach revolves around three steps which is geometric alignment, photometric alignment, and composite view synthesis [17].

In geometric alignment, the first step is to do fish-eye lens distortion correction. After that they estimate a perspective transformation matrix for each of the cameras in order to register them with a ground plane. They work under the assumption that the world is a flat 2D surface.

To do the alignment a calibration pattern is placed in the overlapping region between two neighbouring cameras. They firstly make an initial "guess" of the perspective transformation, after which they use Harris Corner Detection to find the pattern in both images, and then calculate a perspective transform that minimizes the distances between the points in the two images.

Photometric alignment attempts to fix the issue with different illumination between cameras causing the cameras' auto white-balance and auto exposure to make objects look slightly different between cameras. They do this in the overlapping regions, as pixels in these areas are affected by multiple cameras. What they call a tone-mapping is computed for each colour channel in the RGB space, this mapping should minimize the difference between a pixel value from one camera and its spacial counterpart from the other (neighbouring) camera.

In surround view synthesis the geometric and photometric alignments are used to form a composite image. They choose to use alpha-blending to blend the image data in the overlapping regions. The result from the paper can be seen in Figure 3.2(b).



(a) (b)

Figure 3.2: Surround view for embedded systems, (a) the RC car set-up, (b) the end result.

An advantage of this system as can be seen in Figure 3.2(b) is that they have really good results in their simple test scenario.

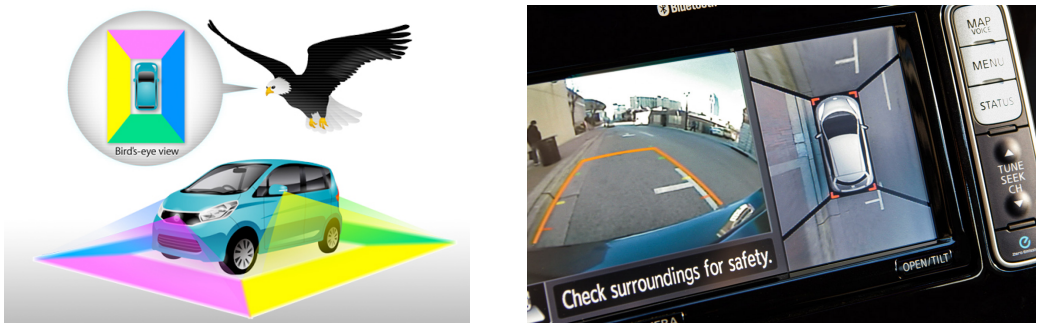
A disadvantage would be that they work under the assumption that the world is flat (as is their test scene), so one would assume far worse results if a non-flat object was inserted into the scene (though they do not show it in the paper).

3.3 Nissan Around View Monitor

In efforts to increase safety for their drivers, many automotive companies continuously work on new technologies to implement in their vehicles. An example of this is Nissan who have created a product they call Nissan Around View Monitor [14]. It solves the same problem as this project aims to, it makes use of four wide-angle cameras mounted on the rear, front, left, and right sides of the car which is similar to the set-up used in this project (although the scale is different).

Their system can show three different things to the driver, a bird's-eye, front, and rear view. They have added other functionalities such as showing where the car is headed with the current position of the steering-wheel through superimposed lines. The bird's-eye view contains small gaps between the cameras, so it does not contain stitching or blending to produce the image. The alignment between the cameras looks good, though objects that are not coplanar with the ground surface will be stretched out and cause misalignment if the object spans multiple cameras.

The system can also do object detection if the car is moving below a certain speed, and will prompt the user by adding a yellow frame to the image. The camera placements, and example of how the views are displayed to the user can be seen in Figure 3.3.



(a) (b)

Figure 3.3: Nissan Around View Monitor, (a) camera positions on the Nissan Leaf and (b) view of the monitor showing front and bird's-eye view.

An advantage of Nissan's system is that it is relatively simple, meaning that it covers the entire 360 degrees though with small gaps between the each neighbouring image which means that they don't loose processing time by stitching the images in the overlapping regions. Another advantage is that they have added extra functionality, such as a simple object detection system, and a secondary front/rear view with superimposed lines showing where the car is going depending on the current position of the steering wheel.

A disadvantage would be that there are some issues with the alignment between the images, and some blurry parts in the image.

3.4 Discussion

Given the results from the systems described above, and their advantages and disadvantages, the implementation(s) for this project can be chosen.

All three systems use fish-eye cameras, and undistort/rectify them using camera calibration or similar (unknown for Nissan), it is an important step, when Computer Vision (CV) approaches are being performed on the images so it is therefore chosen to be the first part of the proposed system for this project.

Given that the goal of this project (as it is with the first two projects listed above) is to create a single bird's-eye view from the four input images, it is important that they are aligned to a shared surface, so they can be combined with each other directly. This is done using a homography matrix to warp the perspective of each input image to align them to a plane.

Lastly the images should be combined into one, preferably the final image does

not have any borders between the four camera images, though from a usability standpoint it will not be vital, as long as the entire area is covered.

Though most of the focus will be on the main approach, a secondary approach will be developed separately from the main approach. This will be based on Computer Graphics (CG) methods in stead of CV. The idea is that each fish-eye images can be projected directly onto a surface in a 3D environment, and if they are placed, shaped and rotated correctly it would produce a somewhat similar result as the main approach. The advantage of this is that once the 3D environment has been set-up it is easy to run it in real-time.

Chapter 4

Design and Implementation

Because of issues that arise from working with fish-eye lenses as seen in [1], it was chosen to implement two different approaches with the same main goal, which is to create a bird's-eye view of the scene. This chapter will explain why these two approaches were chosen, how they work and how they were implemented. The end results of both will be shown in Chapter 5.

Approach 1 is based on Computer Vision (CV) methods, where the images from each camera are to be undistorted using camera calibration, aligned to a shared ground plane, and put together into one composite image. The reason for choosing this approach is that it follows the same basic procedures used by others [10][17][14] and should be able to create a 360 view of the car's surroundings with reduced effects of fish-eye lens distortion

Approach 2 is based on Computer Graphics (CG) methods, where the images from each camera are directly applied as a texture to a spherical surface, with a camera directly above observing the scene. This approach was chosen because it can potentially create the full 360 view in a quite simple way.

It is to be expected that approach 1 will create a scene that looks more natural as it makes use of undistortion, minimizing the effect of radial distortion from the fish-eye lenses, but the image quality will probably be lessened, as the images are warped multiple times. Approach 2 can make the fish-eye effect smaller as well by the use of spherical projection, but certainly not to the same extent.

Approach 1 will for each image have to do many calculations (e.g. undistortion, warping, compositing) making it difficult to run it real-time, while approach 2 will have no issues running real-time once the scene is set up correctly.

Both approaches are inspired by [14] (see Section 3.3), because this is a useful

product for the user, even though it is not perfect. Approach 1 is largely inspired by the approach described in [10] (see Section 3.1)

The two approaches will both be explained, by first describing the idea behind it, then an explanation of how it was implemented, followed by a short discussion on the result.

Both of the approaches described here are making use of image streaming, which makes use of the command line code described in Appendix A, the flow of this can be seen in Figure 4.1.

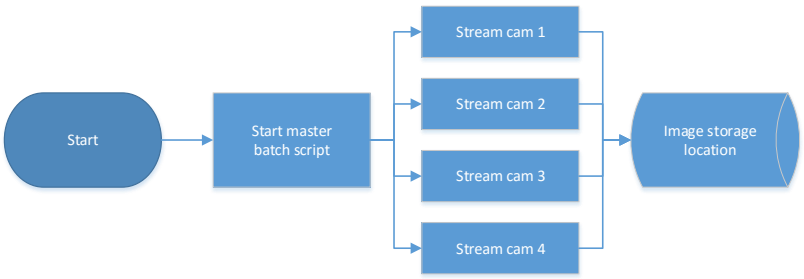


Figure 4.1: Flow of image streaming.

4.1 Approach 1: Planar Alignment and Composite Generation

This approach follows some similar approaches to [10], [17], and [14], which were described in Chapter 3.

4.1.1 Design

The idea of this approach is to create a bird's-eye view of the car where the driver is able to all 360 degrees around the car. The flow of operations can be seen in Figure 4.2. This section describes the full idea of the approach, though not all steps were possible to implement, which will be seen further down.

Firstly the images are undistorted using intrinsic camera parameters to make them from fish-eye images to perspective images. Then each image is aligned to a ground-level plane, so they all share a common plane. Finally the images should be stitched/combined into one image.

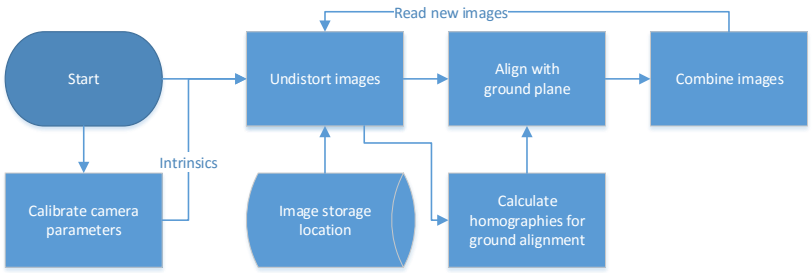


Figure 4.2: Flow chart showing the different operations done in this approach.

4.1.2 Implementation

In order to do the camera calibration, the C++ library OpenCV (also available for other programming languages) is used, as it has all the functionalities needed to do this.

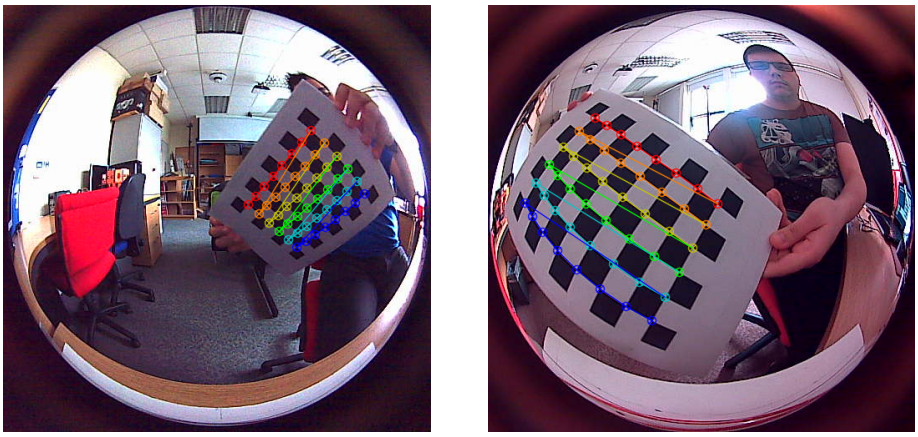
As explained in Section 2.2, a set of images of a calibration pattern at different positions and angles must be used. From each of these images, the corners within the pattern can be found, these are called image points. OpenCV has a function to create an array of points given an image containing a calibration pattern which can be seen in Listing 4.1.

```
127     found = findChessboardCorners(grey, boardSize, imgp);
128     if (found == true)
129     {
130         cout << "found corners in " + camera_string + " img" << endl;
131         cornerSubPix(grey, imgp, Size(5, 5), Size(-1, -1), TermCriteria(
                CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 40, 0.001));
132     }
```

Listing 4.1: Locating chessboard corners.

In line 127, the boolean `found` returns true when the chessboard was found in the image, `grey` is the grey-scale input image, `boardSize` is the number of corners in the pattern on each axis, in this case (8,6) and `imgp` is the image points that are found in the image. The corners in the image are found on pixels, so the second function on line 131 improves the corners by finding them on a sub-pixel level. Two examples of found points can be seen in Figure 4.3, where the corners are marked with colours going from red to blue.

Then a corresponding set of object points are created which should be the same for all images, these points are in 3D, but can be set to 0 on the z-axis. The points are



(a) (b)

Figure 4.3: Example images showing the found chessboard corners.

$(0,0,0),(0,1,0),(0,2,0),\ldots,(1,0,0),(1,1,0),(1,2,0),\ldots,(B_{width}-1,B_{height}-1,0).$

Creating a vector containing image points from all images, and a separate vector for object points, these can be used to calibrate each camera which can be seen in Listing 4.2.

```
116  float rms = fisheye::calibrate(objectPoints, imagePoints, image_size,
         cam_matrix, dist_coeffs,
117      rvec, tvec, fisheye::CALIB_RECOMPUTE_EXTRINSIC | fisheye::
             CALIB_CHECK_COND | fisheye::CALIB_FIX_SKEW);
```

Listing 4.2: OpenCV fish-eye camera calibration function.

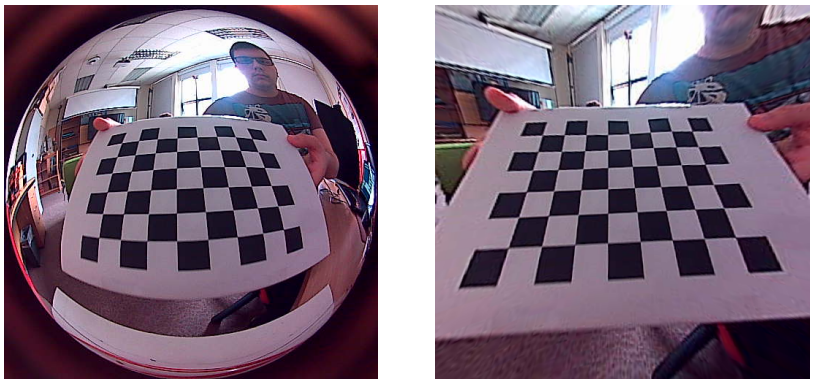
This function takes the object points, image points and size of the original image as input, and outputs a camera matrix which contains intrinsic parameters, distortion coefficients, and lastly rotation and translation vectors for each of the images.

```
102      fisheye::undistortImage(img, dst, cam_matrix, dist_coeffs,
             cam_matrix, Size());
```

Listing 4.3: OpenCV fish-eye undistortion function.

The function in Listing 4.3 undistorts the input image `img` and outputs an undistorted image `dst` using the previously calculated camera matrix and radial distortion coefficients. An example of this can be seen in Figure 4.4.

During implementation it was found that the undistorted image had lost too much of its original FOV, meaning that there was no overlap, so the result would not cover the entire 360 degrees. Alternatives to traditional camera calibration was looked into, but it was found that because these other approaches only guessed



(a) **(b)**

Figure 4.4: Undistortion using camera calibration, (a) input fish-eye image, (b) output undistorted perspective image.

how the image should be rectified, and does not model it like camera calibration, it did not give proper results.

Luckily it was discovered that OpenCV has extra functions for this, which can be seen in Listing 4.4.

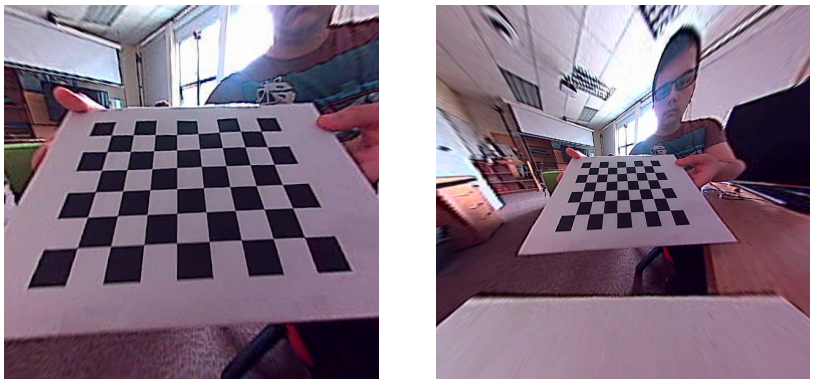
```
120    new_cam_matrix = getOptimalNewCameraMatrix(cam_matrix, dist_coeffs,
           image_size, 1.0, Size());
```

Listing 4.4: Estimating the "new" camera matrix.

This line attempts to estimate what the secondary camera matrix should be, as could be seen in Listing 4.3 `cam_matrix` was used twice. Instead the secondary camera matrix should be substituted instead of the second usage. However because of the heavy distortion the function in Listing 4.4 could not properly estimate values for `new_cam_matrix`, and therefore it was chosen to change them by hand instead. As mentioned earlier a camera matrix contains four main values, two for focal length and two for optical centre. The centre should be the same as the original `cam_matrix` but the focal length should be adjusted so that the resulting undistorted image appears more “zoomed out”, though it does become more noisy and blurry the more it is “zoomed out”. An example of the output making use of the `new_cam_marix` can be seen in Figure 4.5.

it can be seen that especially away from the centre the image still looks very distorted, but the FOV is better which is more important, and for the most part things do not appear curved in the image.

Now all the images have been undistorted and made into perspective images. Then to align all the images to a shared ground plane, the program needs to warp the

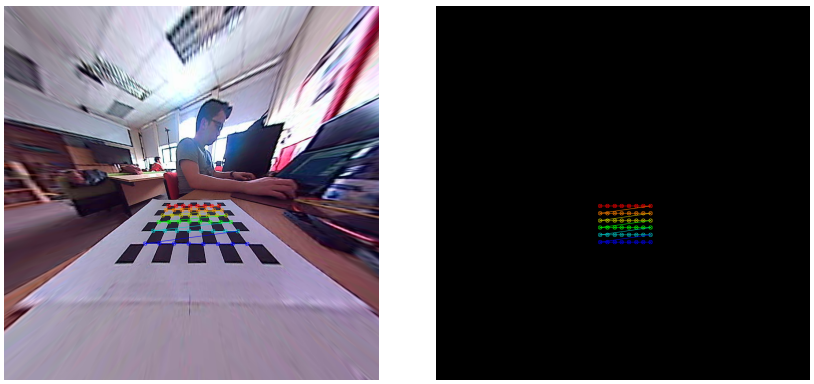


(a) (b)

Figure 4.5: Undistortion using camera calibration, (a) without using the secondary camera matrix, (b) using the secondary camera matrix.

images in perspective so that it appears to have been taken by a camera situated directly above the set-up.

This is done by using the function to find chessboard corners, then creating a new set of points in the same size, but with all distances to neighbours being equal (not diagonally). This can be seen in Figure 4.6.



(a) (b)

Figure 4.6: Two point sets (a) found in the image and (b) new points for ground plane.

From these two point sets a homography matrix can be made that can be used to map from one to the other. This is done using the line of code in Listing 4.5.

```
150     Mat H = findHomography(imgp, newCorners);
```

Listing 4.5: Calculating the homography matrix that maps from one plane to another.

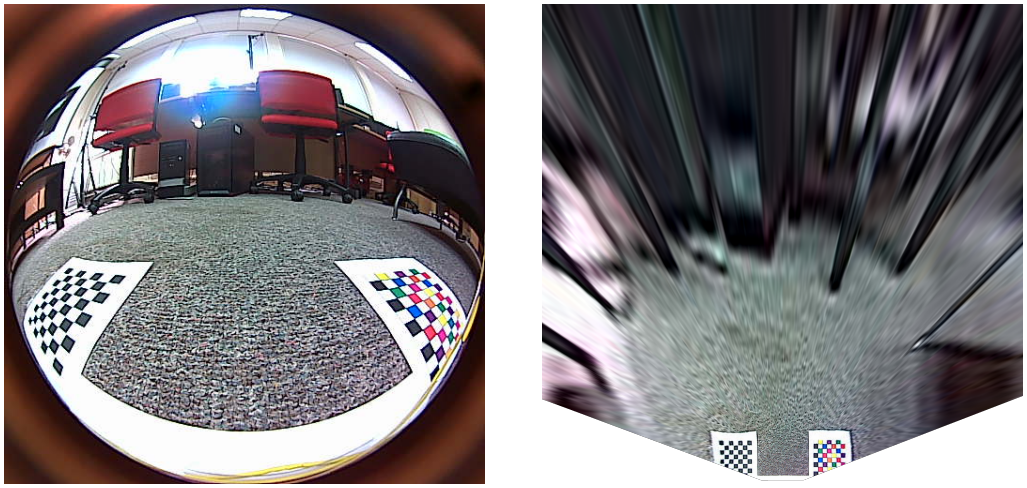
This homography matrix is then used to warp the image to the ground plane as shown in Listing 4.6.

```
164     cv::warpPerspective(src, dst, H, warped.size(), INTER_CUBIC,
            BORDER_TRANSPARENT);
```

Listing 4.6: Warping an image using a homography matrix.

Here the input image `src` is warped in perspective using the homography matrix `H` and saved to the output image `dst`.

Now all four images have been aligned to a shared ground plane, which can be seen in Figure 4.7. It is obvious that the image for the most part looks quite bad, however most of the image can be cut away, as it is too far away from the car to really be relevant in a parking situation.



(a) **(b)**

Figure 4.7: Image from the front mounted camera, (a) input fish-eye image, (b) output undistorted and aligned to ground plane.

From here the initial idea was to stitch the images together into one composite image, but because a lot of the original image quality is lost especially towards the edges where the images overlap, it was not possible to match SIFT features accurately enough. If there were six or eight cameras instead of four, traditional image stitching would likely be more feasible.

For stitching both doing all the steps manually, and using a built-in set of functions from OpenCV was looked into. Specifically the two lines in Listing 4.7 are all that is needed to do stitching.

```
61  Stitcher stitcher = Stitcher::createDefault(try_use_gpu);
62  Stitcher::Status status = stitcher.stitch(imgs, stitched);
```

Listing 4.7: OpenCV stitcher code using default values.

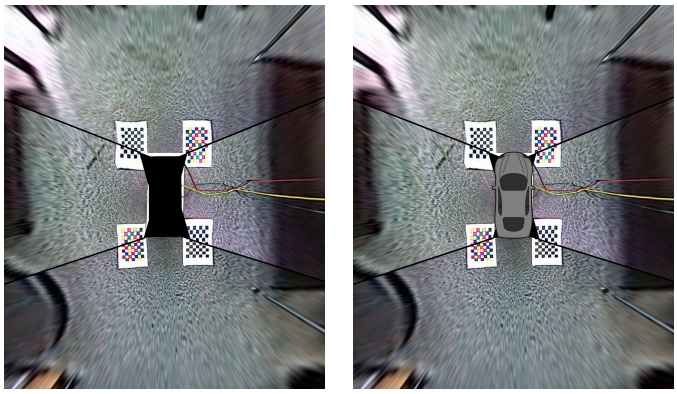
Here the `Stitcher` class is set up to use all its default values, `try_use_gpu` is a boolean, that if true makes the stitcher try to utilize the Graphics Processing Unit (GPU) whenever possible. The second line does all the needed steps of image stitching, it takes the array of images `imgs` (needs to contain two or more images) and returns `stitched`, which is the final stitched together image. It might be necessary to use some non-default methods of the `Stitcher` class, but this was not fully looked into.

Semi-manual stitching was also implemented, but did not perform as well as the fully automated implementation using the `Stitcher` class. The steps of semi-manual stitching revolves around using SIFT to find features, matching them, creating a homography matrix to transform one of the images to the space of the other, warping it using the homography matrix, blending the images into one, and maybe some extra post-processing steps to make the result look better. Most if not all of these things can be done using OpenCV functions.

In this case where the overlap is narrow and the quality in the overlapping region is poor, it was chosen to simply place the four images according to the cameras' real-world position and place the front and rear images on top of the left and right images.

Using a set of images taken with a calibration pattern in each of the overlapping regions, the initial positions of the images were changed in order to align them so the calibration patterns aligned best as possible with no parts repeating (being visible in two camera images). Straight lines have been added between each pair of neighbouring images to make it look more natural when objects moves between them (and change appearance).

As a last step, an image of a car has been added on top of the image, to make it look more like a commercial product, the result of that can be seen in Figure 4.8.



(a) (b)

Figure 4.8: Composite images (a) all four planar aligned images, (b) with an added image of a car on top of it.

4.1.3 Discussion

Flat objects in the scene are aligned very well, however when non-flat objects are added they will be heavily stretched due to the perspective change.

There is an issue with exposure differences between the cameras, making them appear brighter or darker than one another.

Some additional work could be done by adding exposure compensation, but also trying to dynamically warp along the small border lines to make everything align better along them. Something similar is done in [10] along the seams they estimate.

4.2 Approach 2: Direct Spherical Projection using Unity

This approach is an attempt to make a simple approach that works real-time while still being useful for the driver.

4.2.1 Design

The idea of this approach is similar to approach 1, in that they both generate a bird's-eye view of the car's surroundings. It is based on spherical projection, and the flow of operations in this approach can be seen in Figure 4.9.

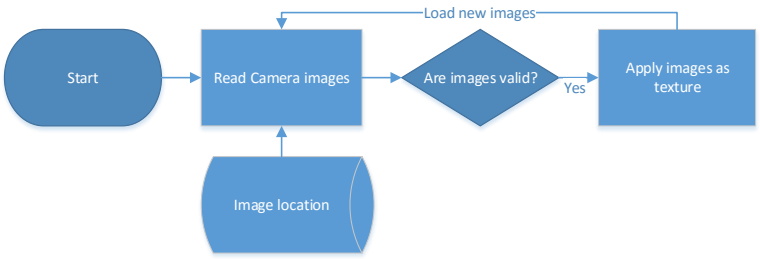


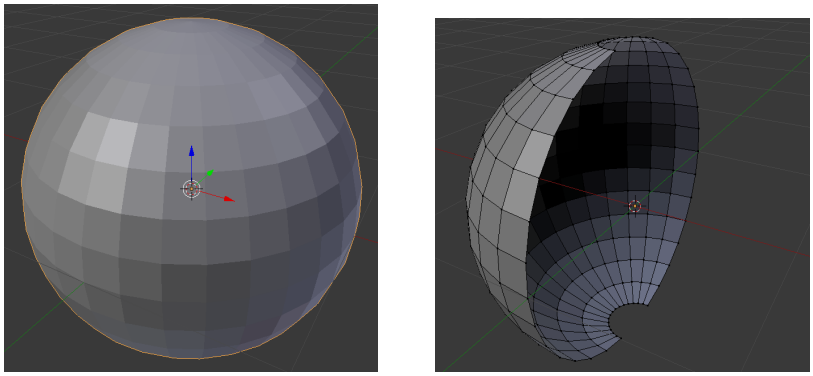
Figure 4.9: General flow of operations in the Unity based approach.

The idea is that once a scene has been created with the required spherical surfaces, that images can just be streamed directly to these surfaces to create the sought after bird's-eye view.

4.2.2 Implementation

Firstly, the wanted surfaces for this is created, the main idea is that they should be spherical so that the shape of them will help undistort the streamed images just from the shape itself. Because Unity does not have any functionality for creating and modelling shapes, Blender was used for this. Two different spherical shapes were created, a half-sphere and a quarter-sphere. The half-sphere will be the basis of this explanation, though the same approach was used for creating the quarter-spheres.

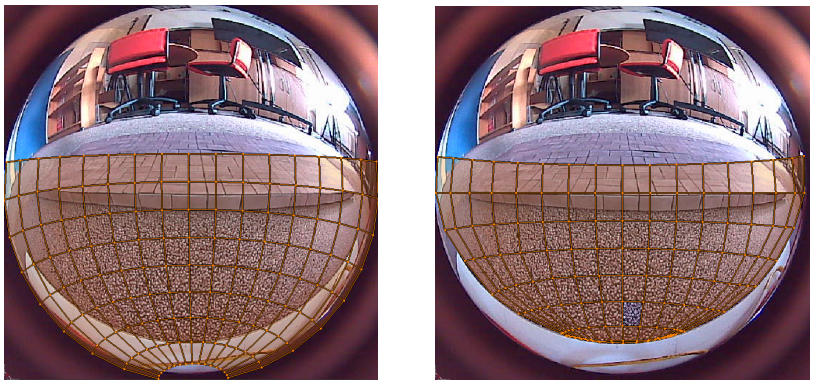
Firstly a sphere is created in Blender as shown in Figure 4.10(a), then half of the circle is removed by marking and deleting their vertices as seen in Figure 4.10(b). Lastly some of the top of the image was deleted, this is because this would be (in the image) the furthest away area from the camera, and therefore not important.



(a) (b)

Figure 4.10: Sphere created in Blender (a) full sphere, (b) half of the sphere deleted.

Then the texture mapping of the half-sphere must be fixed, as can be seen in Figure 4.11(a) if it is left with standard mapping, it will keep the bottom part of the image still there, which does not contain relevant information (in this case parts of the set-up, in a real-world case it could be the side of the car itself). After moving the vertices in the texture mapping, this area will automatically be excluded from the texture when applying it (see Figure 4.11(b)).



(a) (b)

Figure 4.11: Texture mapping (a) directly onto half-sphere, (b) corrected texture mapping.

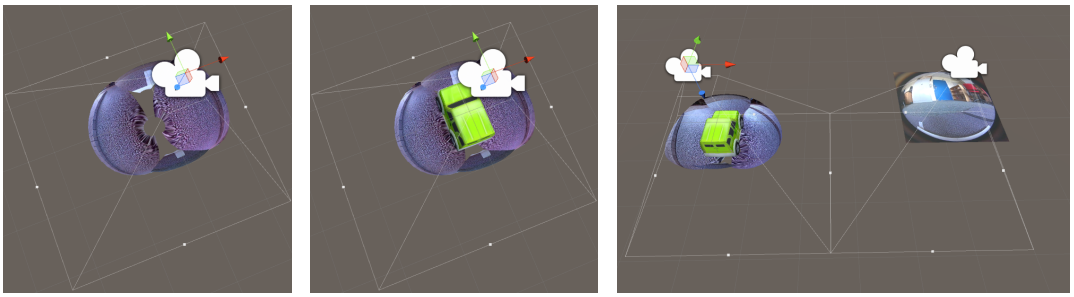
When these surfaces have been created, the scene can be set up. To do this, two of each of the sphere cut-outs are placed according to how the cameras are positioned on the physical set-up. Then a script is created that is able to read the image and apply it as a texture, this can be seen in Listing 4.8.

```
1  ï»¿var url = "file://C:/images/left/image.jpg";
2
3  function Start ()
4  {
5      GetComponent.<Renderer>().material.mainTexture = new Texture2D(4, 4,
           TextureFormat.DXT1, false);
6      while(true)
7      {
8
9          var www = new WWW(url);
10         yield www;
11
12         if (www.texture.width == 512 && www.texture.height == 512)
13         {
14             www.LoadImageIntoTexture(GetComponent.<Renderer>().material.
                   mainTexture);
15             www.Dispose();
16             www = null;
17             Resources.UnloadUnusedAssets();
18         }
19     }
```

Listing 4.8: Reading the camera stream and apply it as texture.

The first line specifies the location of the image, then in line 9, the program starts downloading from the location, yield on line 10 specifies that it will wait until it is done downloading, note that the www module for Unity is made for accessing webpages, but can also access data on the computer's drives. The if statement checks if the image is of the correct size, which in this case is 512x512, this is needed because the stream occasionally produces empty/invalid images. If the image is of a valid size, it is loaded into the texture of the object the script is attached to. The last three lines inside the loops removes the image from the RAM, if this is not done, the RAM will overflow and the program will crash.

To get the textures to be streamed to the inside of the spheres, the normals are flipped. The camera should be moved so that it is placed directly above the spherical surfaces, and pointing downwards which can be seen in Figure 4.12. Then to make it look more like the real world, a car is added to the scene, a model of this was simply found in Unity's own Asset Store.



(a) (b) (c)

Figure 4.12: The unity scene (a) with four spherical surfaces, (b) with an added car, and (c) with an extra camera and plane.

To add to the usability of the system it was chosen to show a secondary view to the user of the rear camera. This was simply added by having a plane with the script to read from the rear camera image location and, similarly as before, adding a facing down camera directly above it. The result of this can be seen in Figure 4.12. The final view from the cameras can be seen in Figure 4.13.

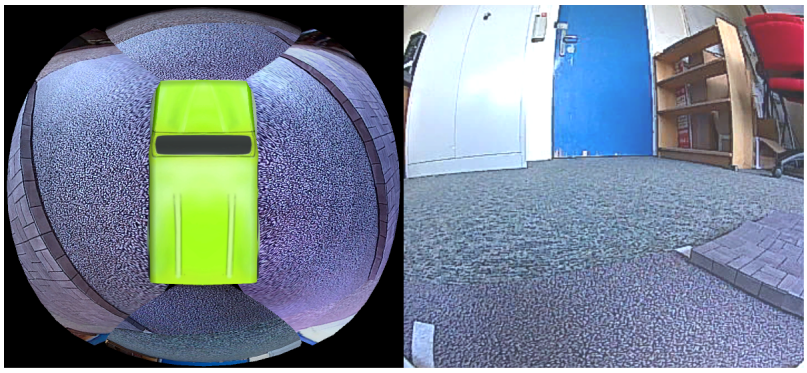


Figure 4.13: The view from the two cameras in the unity scene.

4.2.3 Discussion

This approach has some issues with mis-alignments and objects not transitioning smoothly between camera images and the image still look distorted, but the approach can easily run real-time, and if the issue described in Appendix A with many of the streamed images being empty/invalid was solved, the results would be very usable in a real-world set-up. More work could be done to better the alignment between the camera streams in the scene, but because of the difference in viewing direction and placement, the result will likely never be close to perfect in this rather simple solution.

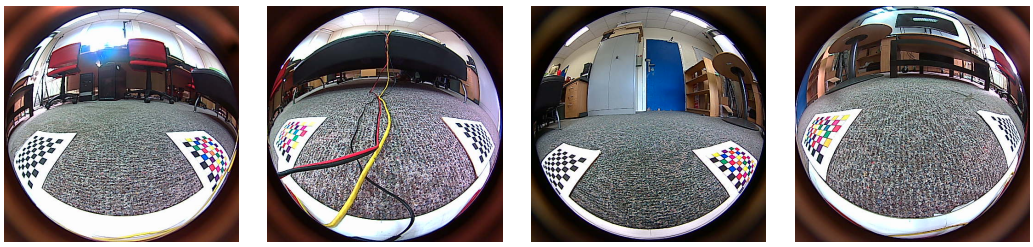
Chapter 5

Results

In this chapter the results from the two approaches described in Chapter 4 will be presented. These results are mostly visual, and they will be used to assess the usability and how realistic the scene is represented in the results. The resulting bird's-eye view will be compared to each other, along with an image taken with a standard camera above the set-up, and lastly with results from the related projects [10],[17], and [14] which were described in 3.

5.1 Test Scenario 1

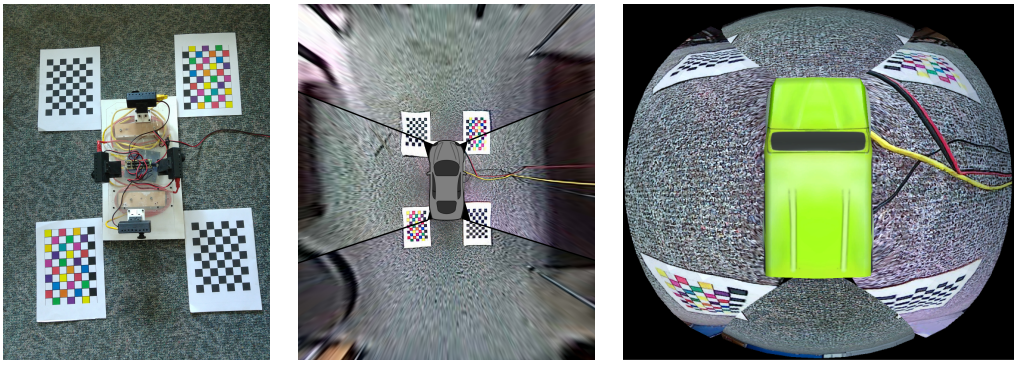
The first scenario is different from the rest, as the set-up is not placed in a realistic scene, but with calibration patterns in the overlapping regions instead. This was used to verify the alignment of the four images with easily recognisable patterns, making it straightforward to see the performance of the system in a completely flat scenario. The four input images can be seen in Figure 5.1.



(a) (b) (c) (d)

Figure 5.1: Scenario 1 from (a) front, (b) right, (c) rear, and (d) left.

The result is compared to an image taken with a standard camera facing downwards on the set-up which can be seen in Figure 5.2.



(a) (b) (c)

Figure 5.2: Bird's-eye view from (a) standard camera, (b) using planar alignment approach, and (c) direct texture mapping approach.

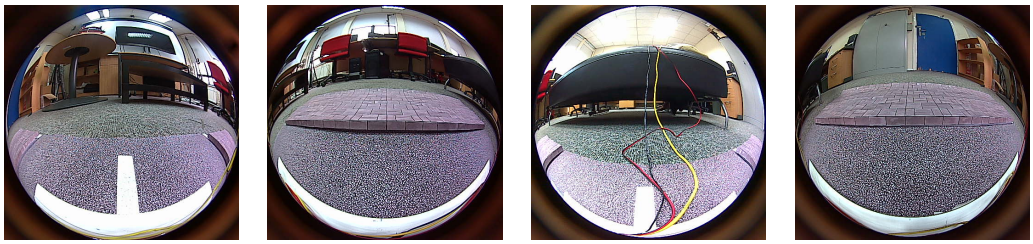
In this Scenario the results from the planar alignment look quite good. Even though the image gets quite blurry towards the edges the alignment is really good with no objects occurring twice. This is the simplest scenario to place the set-up in, as there are no non-flat objects close to the cameras.

The results from the direct texture mapping has some issues with mis-alignments between the images but it does keep more of the details from the original input images than the planar alignment approach because there is no change in perspective applied to the images. The images appear curved as well, making it look as though the car is on a hilltop.

5.2 Test Scenario 2

For the rest of the scenarios, the set-up is placed on a printed surface mimicking a road, with pavement on both sides which is elevated to a height that should be realistic in the small-scale scenario.

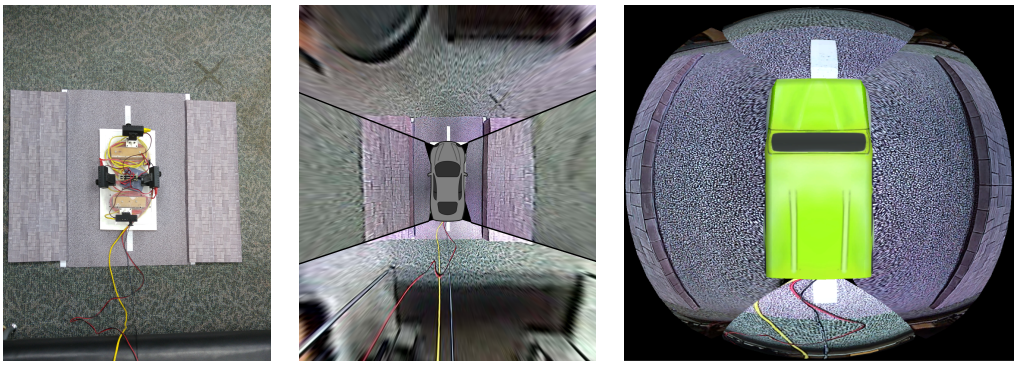
In scenario two, the set-up is placed in the middle of the road where the only non-flat objects in the close range is the pavement. This is a very simple scenario as well, because the small elevation will not be effected a lot by the planar alignment. The input images for the scenario can be seen in Figure 5.3.



(a) (b) (c) (d)

Figure 5.3: Scenario 2 from (a) front, (b) right, (c) rear, and (d) left.

The results from the two approaches can be seen in Figure 5.4.



(a) (b) (c)

Figure 5.4: Bird's-eye view from (a) standard camera, (b) using planar alignment approach, and (c) direct texture mapping approach.

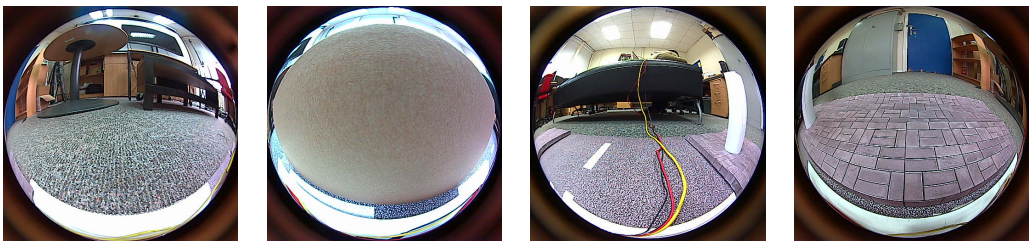
In this scenario the result from the planar alignment approach looks quite good, with the road and pavement having good alignments between the neighbouring images. The outer parts of the pavement on both sides does look quite blurry which is an issue, but given that the proposed use case for this system is for parking assistance, it might be out of the range that is relevant in that situation.

For the direct texture mapping, the alignment looks fine between the side cameras and the bottom camera, whereas the alignment is a bit off with the front camera.

5.3 Test Scenario 3

In the rest of the scenarios, two non-flat objects have been added to the scene. A cardboard box which is roughly the size of the car/set-up and a paper cylinder mimicking a pedestrian.

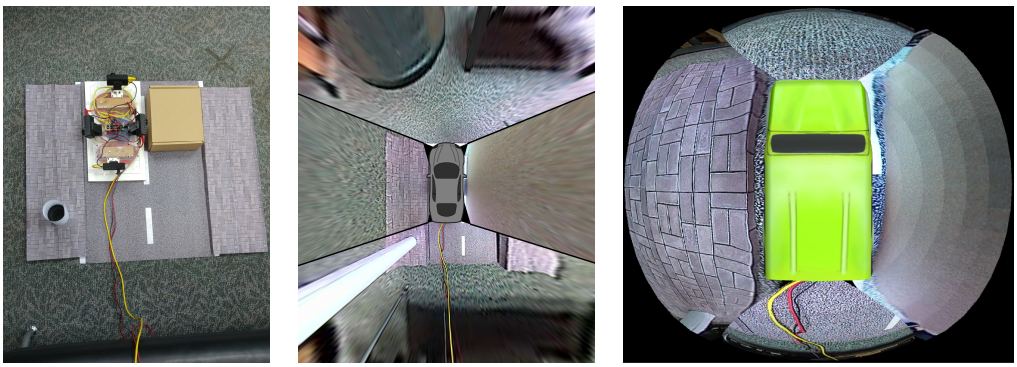
In the third scenario, the box is placed directly next to the set-up's right-hand side only about a centimetre from the right camera, while the cylinder is placed in the bottom-left on the pavement. The input images for the scenario can be seen in Figure 5.5.



(a) (b) (c) (d)

Figure 5.5: Scenario 3 from (a) front, (b) right, (c) rear, and (d) left.

The results from the two approaches can be seen in Figure 5.6.



(a) (b) (c)

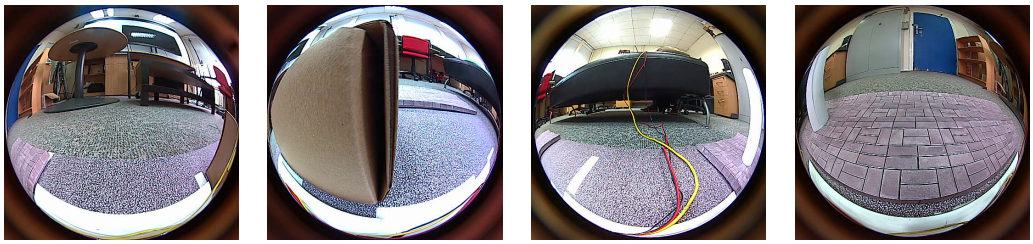
Figure 5.6: Bird's-eye view from (a) standard camera, (b) using planar alignment approach, and (c) direct texture mapping approach.

In this scenario the drawback to any system like this is very apparent. Whenever a non-flat object is in the scene and close to one or more of the cameras, it will get stretched almost beyond recognition. Because the box was so close to the right camera it covers most of what the camera can see. The alignment looks good for the most part but it is hard to judge if the right camera image aligns with the front and back camera images, because of the box taking up all the view of the right camera.

The result looks poor for the direct texture mapping approach with bad alignment between the cameras.

5.4 Test Scenario 4

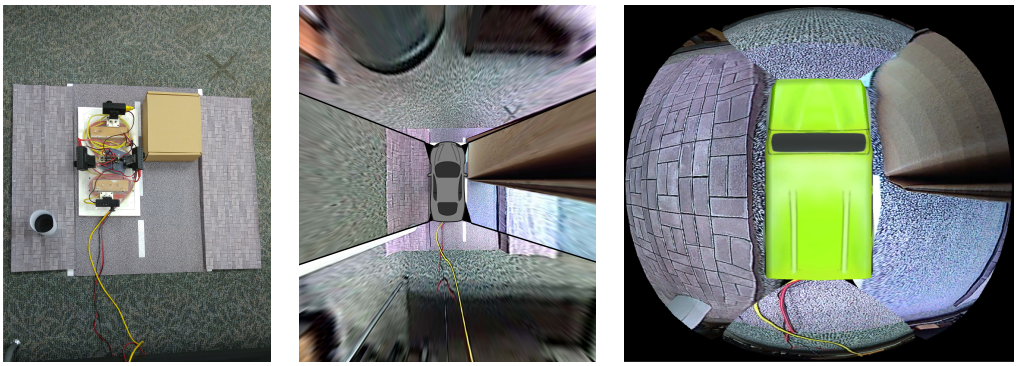
The fourth scenario is close to the same as scenario three, though here the set-up has been moved down so that the right camera is only covered by the box in half of its vision. The input images for the scenario can be seen in Figure 5.7.



(a) (b) (c) (d)

Figure 5.7: Scenario 4 from (a) front, (b) right, (c) rear, and (d) left.

The results from the two approaches can be seen in Figure 5.8.



(a) (b) (c)

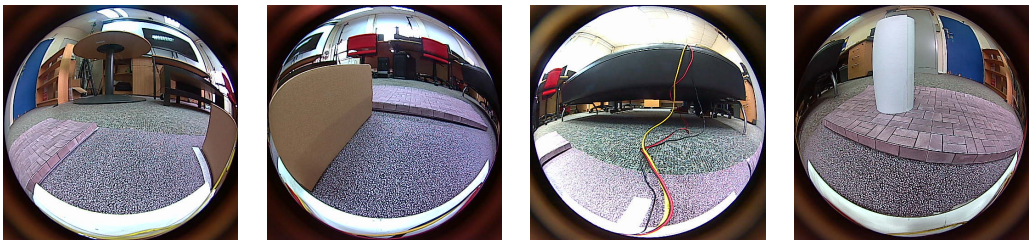
Figure 5.8: Bird's-eye view from (a) standard camera, (b) using planar alignment approach, and (c) direct texture mapping approach.

In this scenario it is clear that when a non-flat object covers the view of two of the cameras, as the box does here, the alignment will not be as good with the current approach. Another approach might be to dynamically align the images across the line between them in a fashion similar to what they do in [10].

Again in this scenario the alignment is off for the direct texture mapping approach, but an advantage of this approach is that the non-flat objects do not look as distorted as in the planar alignment approach, because the perspective is not changed in that way.

5.5 Test Scenario 5

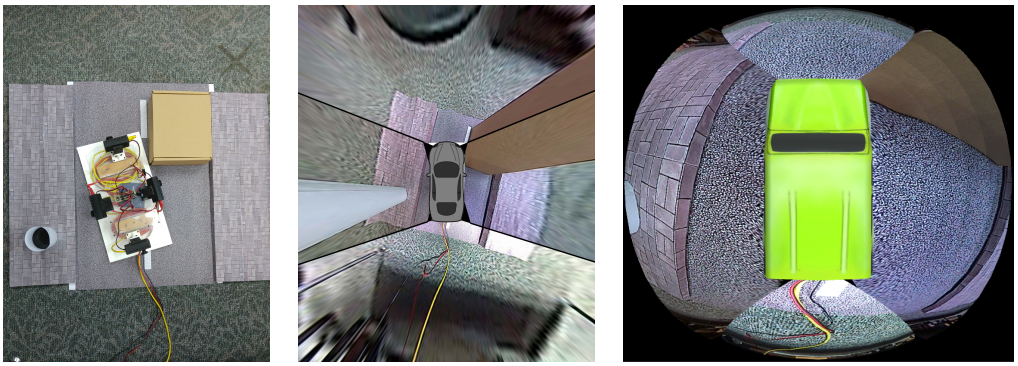
In the fifth scenario the set-up has been moved down and turned slightly so that it is no longer placed parallel to the road. The input images for the scenario can be seen in Figure 5.9.



(a) (b) (c) (d)

Figure 5.9: Scenario 5 from (a) front, (b) right, (c) rear, and (d) left.

The results from the two approaches can be seen in Figure 5.10.



(a) (b) (c)

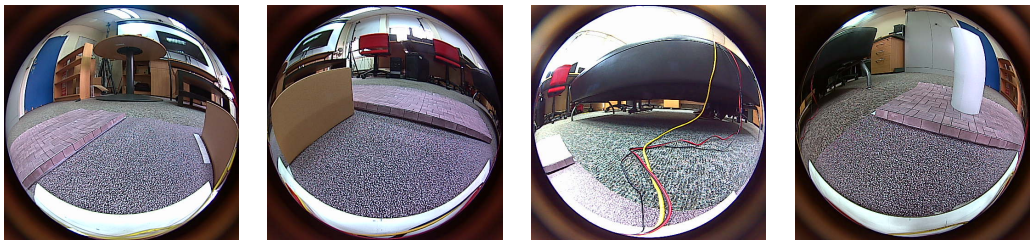
Figure 5.10: Bird's-eye view from (a) standard camera, (b) using planar alignment approach, and (c) direct texture mapping approach.

In this scenario the planar alignment approach has issues with non-flat objects, although the alignment along the road and pavement looks quite good.

The alignment using the direct texture mapping approach is for the most part poor, especially for the front camera.

5.6 Test Scenario 6

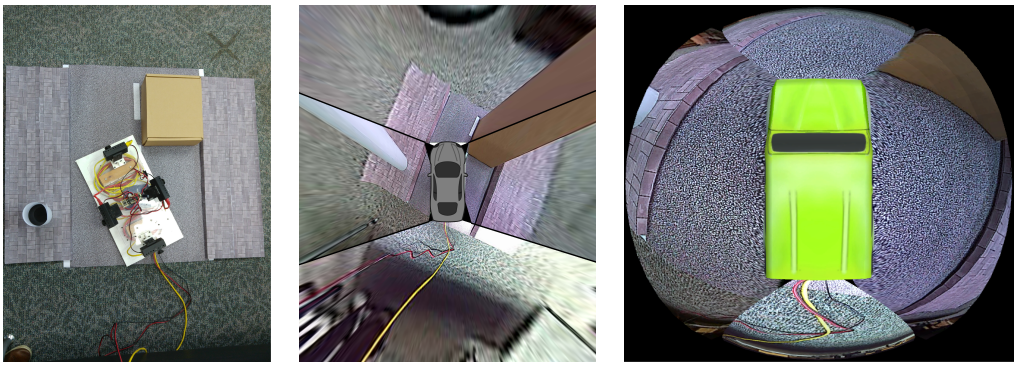
In the sixth scenario the set-up has been moved and turned further. The input images can be seen in Figure 5.11.



(a) (b) (c) (d)

Figure 5.11: Scenario 6 from (a) front, (b) right, (c) rear, and (d) left.

The results from the two approaches can be seen in Figure 5.12.



(a) (b) (c)

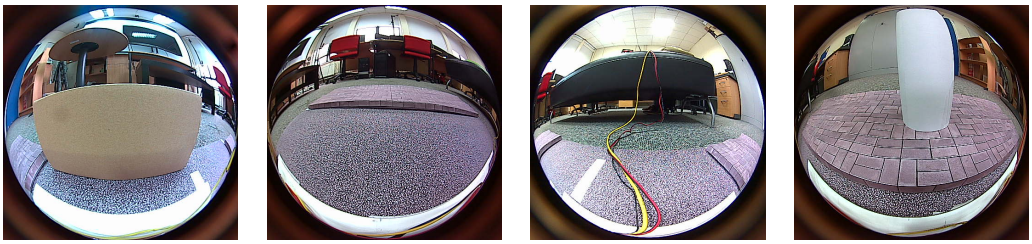
Figure 5.12: Bird's-eye view from (a) standard camera, (b) using planar alignment approach, and (c) direct texture mapping approach.

In this scenario there is another example of non-flat object causing issues for the planar alignment approach. The cylinder gets stretched out, and because of the difference in viewing direction of the front camera to the left, the cylinder looks as if it disappears in the front image, even though (as can be seen in Figure 5.11(a)) it is not in view of the front camera. The alignment of the road and pavement is good here as well. It is obvious that there is an issue with exposure differences between the cameras, this can be seen in the other scenarios as well, but is more visible here, especially in the bottom-right corner where the back camera looks over-exposed compared to the cameras that it neighbours up to.

In this scenario the alignment using the direct texture mapping approach is bad, with none of the neighbouring images having anything close to a smooth transition.

5.7 Test Scenario 7

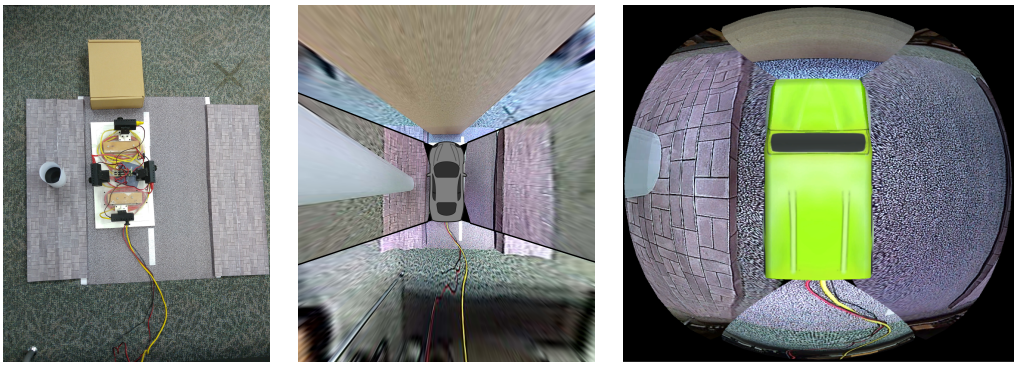
In this scenario the box is placed directly in front of the set-up while the cylinder is placed to the left near the left camera. The input images for the scenario can be seen in Figure 5.13.



(a) (b) (c) (d)

Figure 5.13: Scenario 5 from (a) front, (b) right, (c) rear, and (d) left.

The results from the two approaches can be seen in Figure 5.14.



(a) (b) (c)

Figure 5.14: Bird's-eye view from (a) standard camera, (b) using planar alignment approach, and (c) direct texture mapping approach.

Here because there is a bit more distance to the box, it does not look as bad as it did in scenario 3. But here again the exposure differences is quite obvious, the left and right images look similar, the front looks to have a blue tint to it, while the rear camera is slightly over-exposed.

Again in this scenario for the direct texture mapping approach, the alignment looks a lot better in the lower part of the image, while the top part is a bit off, though this seems to be the scenario where the result from this approach looks best.

5.8 Comparison to Related Work

Firstly the results will be compared to the commercial product from Nissan called Surround View Monitor (see description in 3.3), the images from their system can be seen in Figure 5.15.

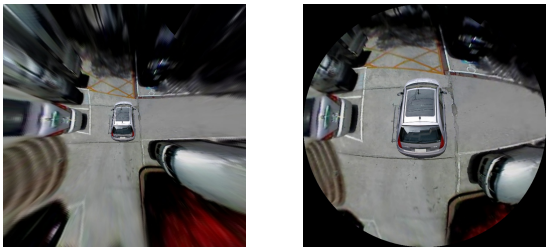


(a) **(b)**

Figure 5.15: Example frames from Nissan's Surround View Monitor. Note that the dog that appears on the right in (a) looks vastly different when it moves to the front in (b) [13].

It can be seen here that the results from this project produces similar, but slightly better results than Nissan's system. They also have issues with non-flat objects, and them changing appearance when moving between cameras because of the large change in viewing direction.

Secondly the results from the project are compared to the results from papers with similar goals. The first is from [10] which has the title Bird's-Eye View Vision System for Vehicle Surrounding Monitoring (see description in 3.1), images of their results can be seen in Figure 5.16.



(a) **(b)**

Figure 5.16: Result from [10]. (a) Without and (b) with a virtual fish-eye effect added.

It can be seen that the results from this project is not quite up to the same level as the results they got in [10]. They were able to create an image without visible overlaps, and without objects changing appearance between neighbouring cameras. This was achieved by dynamically warping the images across the seam between them to align them properly. Using six cameras also helped reduce the effect that the difference in viewing angles can have.

The last related project results to compare with is from [17] with the title A Surround View Camera Solution for Embedded Systems (see description in 3.2), images of the results they obtained can be seen in Figure 5.17.



Figure 5.17: Result from [17]

While their result looks good, with no visible gaps, and good if not perfect alignment, it should be said that they only show results in an entirely flat scenario, and it is unknown how it would perform with a non-flat object added to the scene, though one could guess that it will be severely distorted and might make the alignment poorer if it is placed between two cameras.

From the results shown in this chapter, it is clear that the proposed system have its advantages and disadvantages. The advantages are that the alignments of flat or close to flat objects are quite good. In the scenarios shown above there is no repeating objects near the set-up in the results. Close to the entire 360 view around the car is visible to the user, with only small gaps added to show where the cameras intersect.

A disadvantage is that non-flat object cause issues with over-stretching and misalignment if they cross multiple cameras. This is to some extent a problem that is inherent to the problem itself, and will always be present because the images will always have to be aligned to a common surface, meaning that the perspective of the original images will have to be changed. In [10] they attempt to minimize the effect by trying to model the flat and non-flat objects separately. Another disadvantage is that there are issues with exposure/white-balance differences between neighbouring images. This problem could be solved relatively easily by either doing global exposure correction or changing it in the transitional areas between each neighbouring pair.

The results themselves are comparable to Nissan's commercial product. It offers the same general usefulness to a driver as the system proposed in [10] though the final result does not look as good.

Chapter 6

Discussion

In this chapter some ways to improve or move the project forward will be explained. After this there will be an overall conclusion on the results and the outcome of the project as a whole.

6.1 Future Work

In this section some ways to improve the system described throughout the report will be mentioned.

A small improvement to the system would be to correct the difference in exposure between neighbouring cameras. This could be done both on a global scale by histogram fitting, or be done locally in the transition area between the two neighbouring images. It was chosen to skip this as it was not seen as a vital inclusion.

More cameras, as shown in [10], using six cameras instead of four would result in larger overlaps between neighbouring images in the undistorted perspective images. The main reason that image stitching was difficult to implement in this project was that the image quality of the calibrated images was poor towards the edges and that objects that overlap look very different because of the 90° difference in viewing angle. Using more cameras could help make stitching more practical. Though arguments could be made that the small gaps between the cameras are unimportant to the user.

The cameras used for this project are low resolution, running only at 512x512 or 0.26 megapixels (only about 0.2 megapixels if one takes out the areas with no relevant data outside the image circle). If the images had a higher resolution with

more details it would not only make the result look better, but might make it easier to stitch the images as well.

An obvious next step is to mount the cameras either on a real car or a set-up on full scale. It would be interesting to see how the system would perform in a real-world scenario.

6.2 Conclusion

In the problem description (see Section 1.1) it was stated that the goal of the project is to make it possible for the user to see everything surrounding the car.

From the results in Chapter 5 from the approach described throughout this report, it can be seen that the user would be able to see everything around the car, and even though non-flat objects such as other cars would be severely stretched/distorted, it would still be very useful to a driver, as it covers all the blind spots he/she would otherwise have when using only windows and mirrors (see Figure 1.1 on page 2).

There is still a slight gap between the four camera images, though this is seen as inconsequential, as it is such a small gap that if an object was in that area, it would also cover one or both of the cameras it is between.

Because the goal of the project was reached, it can be said as a conclusion that the results are good enough. Though there is room for improvement, some of still-present issues are either, in case of resulting image quality in some areas, because of hardware limitation, or in case of non-flat objects getting stretched out, an inherent issue present in a project such as this where perspective change is an essential part.

Bibliography

[1] Madhav Bhaku. *Panoramic View Synthesis and Image Fusion*. 2015.

[2] Gary Bradski and Adrian Kaehler. "Learning OpenCV". In: [Accessed on: 10/12/2015]. 2008. Chap. 11, pp. 370–404. URL: http://cdn.oreillystatic.com/books/9780596516130/ch11.pdf.

[3] Matthew Brown and David G. Lowe. "Automatic Panoramic Image Stitching using Invariant Features". In: (2007). [Accessed on: 28/05/2016]. URL: http://matthewalunbrown.com/papers/ijcv2007.pdf.

[4] *Clouds over the port at Takeshiba pier, Tokyo*. [Accessed on: 14/04/2016]. 2007. URL: https://www.flickr.com/photos/heiwa4126/1887527159.

[5] Mike Day. *Extracting Euler Angles from a Rotation Matrix*. [Accessed on: 08/02/2016]. 2012. URL: https://d3cw3dd2w32x2b.cloudfront.net/wp-content/uploads/2012/07/euler-angles.pdf.

[6] Kai-Chen Huang et al. "A 360-degree Panoramic Video System Design". In: (2014). [Accessed on: 28/05/2016]. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6834863.

[7] C. Hughes et al. "Wide-angle camera technology for automotive applications: a review". In: (2008). [Accessed on: 08/04/2016]. URL: http://www.car.nuigalway.ie/documents/chughes_ietautomotivefisheye.pdf.

[8] Kolor. *Understanding Projection Modes*. [Accessed on: 14/04/2016]. 2014. URL: http://www.kolor.com/wiki-en/action/view/Understanding_Projecting_Modes.

[9] Hyungtae Lee et al. *Scale-invariant object tracking method using strong corners in the scale domain*. [Accessed on: 15/04/2016]. 2009. URL: http://opticalengineering.spiedigitallibrary.org/article.aspx?articleid=1089124.

[10] Yu-Chih Liu, Kai-Ying Lin, and Yong-Sheng Chen. "Bird's-Eye View Vision System for Vehicle Surrounding Monitoring". In: (2008). [Accessed on: 11/04/2016]. URL: http://link.springer.com/chapter/10.1007%2F978-3-540-78157-8_16.

[11] David G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: (2004). [Accessed on: 14/04/2016]. DOI: 10.1023/B:VISI.0000029664.99615.94. URL: http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94.

[12] Rithwik Mutyala. *Scale Invariant Feature Transform*. [Accessed on: 12/02/2016]. URL: https://www.eecs.tu-berlin.de/fileadmin/fg144/Courses/10WS/pdci/talks/sift-feature_extraction.pdf.

[13] Nissan. *2014 Nissan Rogue with Around View Monitor*. [Accessed on: 12/05/2016]. 2014. URL: http://nissannews.com/en-US/nissan/usa/channels/us-united-states-nissan-models-rogue/releases/nissan-rogue-crossovers-hot-sales-continue-as-it-readies-for-2016-with-new-features-and-technology/videos/2014-nissan-rogue-with-around-view-monitor?la=1.

[14] Nissan. *Around View Monitor*. [Accessed on: 15/04/2016]. 2014. URL: http://www.nissan-global.com/EN/TECHNOLOGY/OVERVIEW/avm.html.

[15] OpenCV. *Introduction to SIFT (Scale-Invariant Feature Transform)*. [Accessed on: 15/04/2016]. 2013. URL: http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_feature2d/py_sift_intro/py_sift_intro.html.

[16] Shanghai Optics. *Fisheye lens Case Study*. [Accessed on: 15/04/2016]. URL: http://www.shanghai-optics.com/products/1669-2/.

[17] Buyue Zhang et al. *A surround view camera solution for embedded systems*. [Accessed on: 15/04/2016]. 2009. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5406797.

[18] Zhengyou Zhang. "Emerging Topics in Computer Vision". In: [Accessed on: 10/12/2015]. 2003. Chap. 2, pp. 5–43. URL: http://research.microsoft.com/en-us/um/people/zhang/Papers/Camera%20Calibration%20-%20book%20chapter.pdf.

[19] Haijiang Zhu, Jinfu Yang, and Zhongtian Liu. "Fisheye Camera Calibration with Two Pairs of Vanishing Points". In: (2009). [Accessed on: 28/05/2016]. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5190078.

Appendix A

Image Acquisition

Given that this project makes use of a certain set-up, the acquisition of images from the cameras can only be done in one way. The cameras mounted on the set-up are ethernet cameras streaming through UDP (User Datagram Protocol). As this project continues the work in [1], the same program called GStreamer is used to connect to the stream and store images.

Gstreamer is a framework for multimedia that is based on pipelines. A pipeline is essentially sequential software, meaning that the output of one process/function is the input to the next, occasionally some of the processes can be run in parallel. Since GStreamer is made for multimedia these different process could for example be encoding/decoding, filters, etc.

So the way that images are saved using GStreamer is to set up a pipeline, in this case a quite simple one is used, which connects via UDP to a *sink* and stores it somewhere on the PC. The command for this was put into a batch file, that opens the stream, and if the stream terminates it will restart, this can be seen in Listing A.1.

```
1 cd C:\gstreamer-sdk\0.10\x86_64\bin
2 :start
3 gst-launch-0.10.exe udpsrc caps="application/x-rtp, media=(string)video,
    clockrate=(int)90000, encoding-name=(string)JPEG, payload=(int)96,
    framerate=30/1" port=20004 ! rtpjpegdepay ! multifilesink location=C:/
    images/front/image_%%d_front.jpg
4 goto start
```

Listing A.1: Batch file for opening a stream from the front camera.

Where it is important that the settings and port match the transmitter, the program could also be `gst-launch-1.0.exe` if that version is used, there should not be any difference in the functionalities in this simple example though.

A *master* batch file was then created, which starts all four camera streams. A slightly altered version of the code shown in Listing A.1 was made where the only change was location, where the image name was change from "image_%%d.jpg" to "image.jpg". With this change, the cameras only stream one image, which is continuously updated instead of creating a large number of images.

One of the issues encountered when using these cameras is that they occasionally stream images that are empty, and therefore unusable. A non-optimal workaround to this would be to always check if the image is valid for example using the image size, and only updating the image when the image is valid. This will however cause the cameras to be out of sync with each other, as they will independently from each other have empty images at random times, which will cause the resulting bird's-eye view to be confusing to the user.